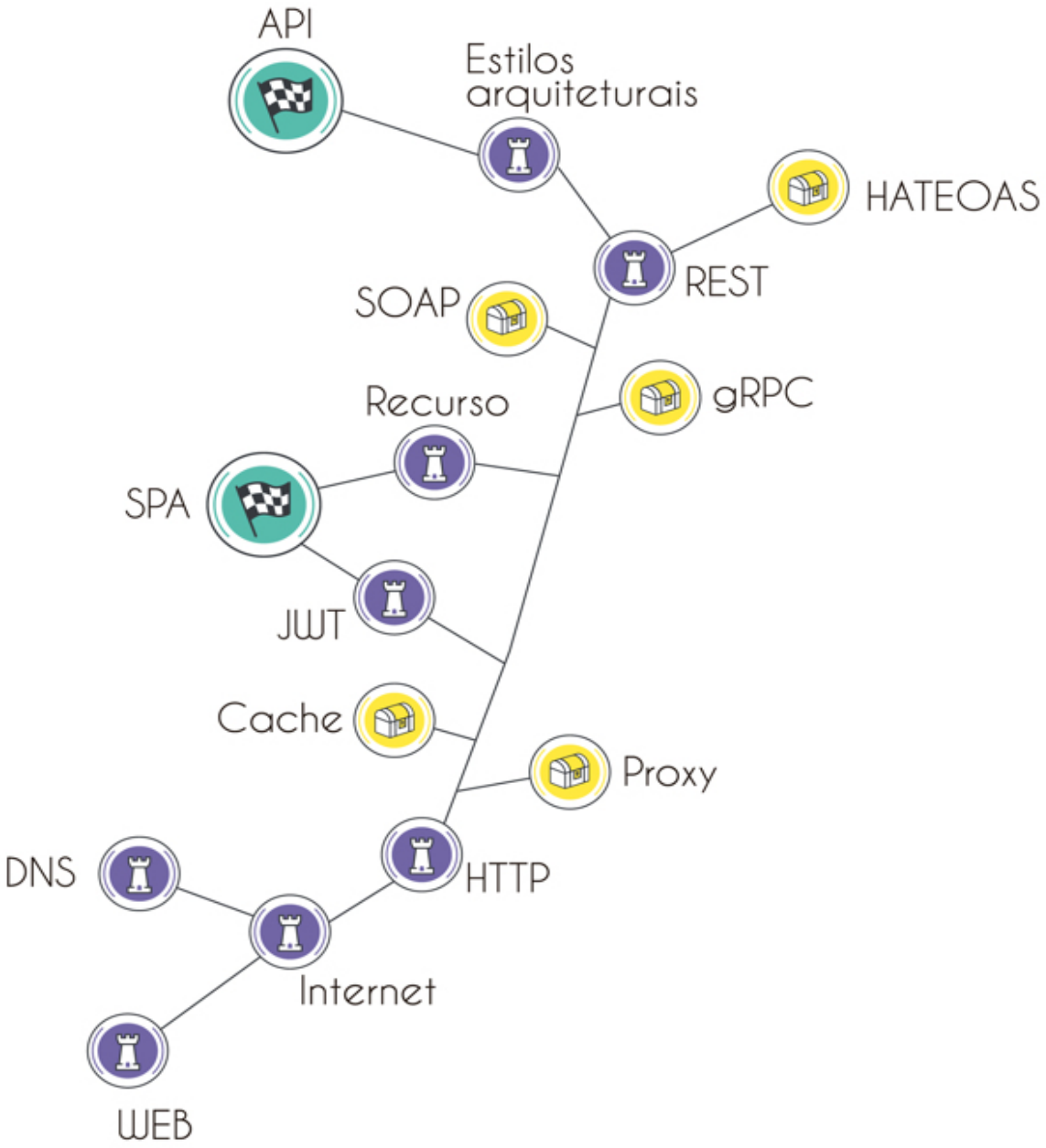
Roadmap back-end

Conhecendo o protocolo HTTP e arquiteturas REST



Casa do Código | alura

VICTOR OSÓRIO

Sumário

- ISBN
- Sobre o livro
- Agradecimentos
- Sobre o autor
- Quem somos e onde vivemos?
  - 1 Tudo o que você pode ser
  - 2 Uma breve história da Internet
- Protocolo HTTP
  - 3 HTTP, o protocolo da web
  - 4 Discutindo o protocolo
  - 5 Implementando o protocolo com Quarkus
- Estilos arquiteturais usando HTTP
  - 6 Estilos arquiteturais e REST
  - 7 Outros estilos para HTTP
  - 8 Próximos passos

ISBN

Impresso: 978-85-5519-297-5

Digital: 978-85-5519-296-8

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Sobre o livro

Quando eu entrei na universidade, o mote do Instituto de Computação da Unicamp era algo como "Computação é arte". Eu não sei o que você entende por arte, mas quando eu penso em uma obra de arte eu penso que ela deve ter várias dimensões estéticas e qualidades subjetivas, logo eu tento que todo o meu projeto seja leve, de fácil compreensão e belo. Isso me leva a procurar padrões, tentar identificar estilos, manter sempre um sistema fácil de ser compreendido. Eu considero meu código e os produtos que ele compõe uma obra de arte.

É óbvio que nem sempre alcançamos a perfeição, na grande maioria dos casos temos um prazo e precisamos fazer o possível para entregar funcionalidades dentro de uma agenda. Mas só entregar funcionalidades o mais rápido possível pode nos levar a adicionar complexidade o que vai impactar no médio ou longo prazo, por isso o equilíbrio é sempre desejável. Equilíbrio é uma das características de uma obra de arte, sempre teremos isso em mente.

Neste livro vamos discutir um pouco sobre desenvolvimento back-end. Ele não é um livro exaustivo, tem muito conteúdo que não foi coberto, mas ele focará no ponto principal que liga um servidor ao seu cliente: o protocolo HTTP.

Uma boa API HTTP deveria ser que nem uma obra de arte. Ela deve ter características e estilo. Os usuários dela devem ter uma boa experiência ao usar. Apresentar uma API com diversas funcionalidades, mas simples, deveria ser o objetivo de toda pessoa que a desenvolve.

Este é um livro introdutório organizado em 3 partes. Na primeira, vamos apresentar o que é o trabalho de alguém que desenvolve um servidor back-end. Depois vou falar sobre como as coisas evoluíram para chegar ao momento em que estamos vivendo, você vai conhecer um pouco sobre a história da Internet e das tecnologias que sempre atuaram detrás das cortinas. Na segunda parte, vamos

falar especificamente sobre o protocolo, com o objetivo de detalhar como ele é definido, como pode ser usado e implementado facilmente. E na terceira parte vamos falar sobre estilos arquiteturais, vamos definir o que é um estilo arquitetural e quais são os estilos que são usados através do HTTP.

O conteúdo pode até ser todo focado em HTTP, mas vamos definir muitos conceitos e trabalhar um pouco do que é arquitetura. Eu espero que você aprenda com este livro e que ele ajude você a ser muito melhor no seu trabalho.

Boa leitura!

Victor Emanuel Perticarrari Osório

Agradecimentos

Escrever um livro não é fácil. Este que você tem em mãos foi escrito no segundo ano da pandemia da COVID-19, ano do qual nós passamos grande parte em isolamento social - e, quanto mais isolamento, menos empolgação temos para os projetos paralelos. Por isso agradeço enormemente a Vivian Matsui pela paciência e por ter aceitado entrar nesse projeto. No começo era um grande livro sobre desenvolvimento back-end que provavelmente demoraria anos a ser escrito, mas ela pacientemente foi podando as ideias até se tornar um livro direto e acessível. Obrigado.

Sou grato também a minha esposa Danielle por ter, por inúmeros domingos, ter sido privada da minha companhia para que eu pudesse focar na escrita do livro. Antônio que teve que entender que o papai estava trabalhando em um domingo e a Catarina que foi gestada no meio dessa empreitada.

E você que está lendo é o motivo deste livro. Se eu consegui ajudar você em alguma coisa, já valeu a pena todo o esforço.

Muito obrigado!

Sobre o autor

Victor Osório é um engenheiro de software formado em Engenharia de Computação pela Unicamp em 2007. Desde 2005 vem trabalhando com Java em projetos de diferentes contextos. Já atuou com desenvolvimento de IDEs para desenvolvimento mobile entre 2006 e 2010. Entre 2012 e 2015, trabalhou com soluções para síntese de fala, reconhecimento de voz e outras soluções em IA relacionadas a fala. E desde 2015 tem trabalhado com microsserviços e sistemas orientado a eventos. Dentre esses vários contextos, desde 2010, tem trabalhado com o desenvolvimento de APIs e busca fazer o melhor design para otimizar o uso de recursos e a facilidade de desenvolvimento.

Piauiense nascido em Teresina, reside em Campinas desde 2002 e evitou trabalhar na capital paulista até 2016, para onde acabou migrando apenas profissionalmente. Em Campinas tinha uma vida "bucólica" aproveitando a efervescência cultural ao redor da Unicamp e dois dos melhores bares do Brasil segundo a MTV na longínqua década de 2000. Mas ao trabalhar em São Paulo teve contato com outro mundo onde meetups e comunidade de desenvolvedores se tornaram eventos mais comuns.

Hoje trabalha calmamente em casa sendo especialista de software na Amdocs, desenvolvendo uma plataforma para processamento de eventos usando Apache Kafka e usa seu tempo livre para tirar dúvidas sobre assuntos aleatórios no Twitter e escrever alguns textos para ajudar as pessoas a produzir melhores códigos e sistemas. Já deu palestras sobre diversidade cultural, filosofia do design de código, Java e Apache Kafka.

Quem somos e onde vivemos?

Como pessoas desenvolvedoras, quem nós somos? O que é esse espaço onde habitamos?

Capítulo 1
Tudo o que você pode ser

Se você chegou aqui eu tenho uma leve suposição sobre você: **Você tem interesse no mundo Back-end**. Bom, muito bom!

Já que tem esse interesse, vou lhe ensinar tudo o que é preciso saber para entrar nessa jornada. Vamos iniciar supondo que você não sabe nada além da sua linguagem ou de alguns *frameworks* - que são bem úteis, mas que podem esconder toda a magia da Internet.

Nosso objetivo é, durante essa jornada, dar as bases para que você amadureça conhecendo, além dos frameworks, os protocolos e arquiteturas envolvidas no seu trabalho do dia a dia. Vamos nos preocupar aqui com as *Hard Skills* (ou seja, além dos protocolos, um pouco das técnicas e estilos arquiteturais), focando em dar as bases para que possa expandir a sua capacidade de resolução e identificação de problemas.

Mas antes de começar, precisamos deixar um pouco a ansiedade de lado e ir construindo passo a passo o conhecimento necessário. Ninguém nasce sabendo tudo e ninguém aprende tudo da noite para o dia. É preciso muito suor para se construir uma experiência. E isso envolve o investimento em tempo.

Espero que ao final dessa jornada você possa ser tudo aquilo que poderia ser no mundo Back-end.

Frameworks e bibliotecas

Hoje, frameworks são personagens centrais no mundo do desenvolvimento de software. Quem não conhece ao menos o nome de um: Spring, Quarkus, .Net, Angular, React, entre outros? Mas como se define um framework? O que diferencia um framework de uma simples biblioteca?

Frameworks são bibliotecas que gerenciam o controle de um código. Ao desenvolver uma aplicação, se você está realmente usando um framework, este deve ter o controle do seu código. Caso contrário, é apenas uma biblioteca. Normalmente os frameworks possuem uma documentação extensa, pois devem detalhar cada ponto de uso, cada detalhe interno. Quem o usa deve conhecer sua dinâmica.

Já bibliotecas são simples porções de códigos que podem ser usadas livremente. Nesse caso, quem usa é responsável pela dinâmica.

Vamos começar pensando em quais são as características básicas que devemos ter? Que mentalidade podemos cultivar para estar sempre crescendo?

1.1 A mentalidade do crescimento

Para começar, podemos definir qual mentalidade precisamos para crescer. Existe uma palavra em inglês para isso, mas, já que falamos português, vamos usar em português mesmo. De todo modo, caso você queira mais dicas, procure por *Growth Mindset*, e filtre bastante o que encontrar.

Para crescermos precisamos alimentar alguns hábitos. Os que eu mais recomento são:

- Estar sempre aprendendo
- Ter critérios bem estabelecidos
- Saber o que é esperado de você
- Entregar sempre um bom trabalho
- Ser mestre em ao menos um framework
- Conhecer os fundamentos da Internet

Estar sempre aprendendo

— E se amanhã surgir um novo framework completamente novo? Como eu fico? Estarei desatualizado?

Essa é uma pergunta realmente válida, é um medo que todos temos. E se um dia acordarmos e nos percebermos ultrapassados? Não pense que isso acontece só com você, todos têm esse receio. Mas há um antídoto para isso! **O bom conhecimento**.

No mundo do desenvolvimento de software é comum focarmos em tecnologias, linguagens e frameworks. Muitas vezes são eles que nos são requisitados para entrar em uma vaga. Vemos lá que pedem Java, Spring, Go, Python, Node.JS, o que é muito para uma pessoa só. Mas... Durante o seu trabalho você vai se deparar com **conceitos**. Estes são esquecidos nos processos seletivos e também ignorados pela maioria dos desenvolvedores.

Nas mentorias que tenho feito, muitos afirmam que sabem construir APIs Rest, mas quando peço para me definir o que é uma API Rest, ou usam uma definição incorreta ou não o sabem. Se você souber o que é uma API Rest, você poderá construir uma mesmo que todos os frameworks desapareçam. Poderá avaliar o seu próprio trabalho a fim de sempre evoluir.

Mas... E se surgir um novo protocolo de comunicação que não usa Rest, nem HTTP etc.? Calma, provavelmente esse protocolo terá uma árvore genealógica dentro da tecnologia. Eles não surgem do

nada, mas sim são construídos com o tempo e reúsam outros conceitos para construir coisas novas. Se você conhecer apenas os frameworks, pode se tornar ultrapassado facilmente, mas se você conhecer os conceitos, não.

Alguns anos atrás, o conceito de desenvolvimento Web estava atrelado a conhecer frameworks que imitavam a arquitetura de aplicações Desktop, então era fácil trabalhar nos dois contextos e intercambiar. Mas hoje isso é impossível. A arquitetura de uma aplicação WEB não mudou, o que mudaram foram os frameworks. Quem conhecia HTTP, HTML e JavaScript pode migrar para a nova realidade.

Nessa jornada vamos aprender vários conceitos e como eles já são usados no nosso dia a dia. Mas coloque um objetivo na sua vida. **Aprender algo novo todo dia**. Assim você nunca estará obsoleto.

Ter critérios bem estabelecidos

— Como podemos classificar algo como bom? Como podemos avaliar se nosso trabalho é bom ou não? Como podemos melhorar a cada dia?

Para responder a essas perguntas, precisamos fazer outra pergunta antes. Como podemos qualificar algo como bom? Não podemos qualificar nada sem antes ter critérios claros e bem estabelecidos.

Ter critério é uma característica bem difícil. Porque muitas vezes ela pode ser qualificada como positiva, mas na maioria das vezes é qualificada como negativa. Soaria negativo se eu dissesse que devemos ser críticos, e, realmente, é negativo. Porque a crítica pela crítica não nos leva a melhorarmos, serve apenas para desqualificar algo. Logo, eu nunca advocaria para que sejamos mais críticos. Então quero propor que sejamos criteriosos.

Ser criterioso é avaliar as coisas cuidadosamente com critério. As palavras "crítico" e "criterioso" são na verdade sinônimos, mas com uma carga de significância totalmente diferente. Em ambas temos

uma pessoa que tem critérios claros para fazer avaliações, mas uma pessoa crítica apenas é alguém difícil de lidar chegando a ser impiedosa em suas avaliações. Porém, uma pessoa criteriosa é cuidadosa em suas avaliações.

Ora, mas como isso pode nos levar a sermos melhor no que fazemos? Primeiro devemos ter muito cuidado ao avaliar o que nós fazemos e os outros fazem. Apenas desqualificar não ajuda em nada. Precisamos conhecer várias abordagens e ponderar o que há de bom e ruim. Só assim podemos ter critérios claros. Se ainda não estiver claro, deixe-me exemplificar: já que desenvolvemos software, como deve ser um bom software?

Há ferramentas objetivas que podem mensurar a qualidade do seu código. Faça uma busca por elas e configure-as em seu projeto. Estabeleça métricas, veja como está a evolução do seu projeto e a sua também. Isso é conhecido como análise estática de código.

Mas há também ferramentas subjetivas. Com essas não podemos simplesmente qualificar algo como bom ou ruim, tudo vai depender do contexto. Por isso, recomendo que você conheça padrões e antipadrões de projetos. Há um mal-estar em relação aos famosos *design patterns*, isso porque muitas vezes eles são usados para criticar negativamente. Vale lembrar que *design patterns* não são uma invenção, eles são uma constatação. São construções comuns em projetos que ao serem identificados são levantados pontos positivos e negativos e os momentos de se usar ou mesmo evitar. Portanto, conhecê-los pode nos ajudar a resolver problemas rapidamente.

Padrões e antipadrões não devem ser um absoluto, devem ser analisados em cada contexto. Alguns desenvolvedores se tornam fundamentalistas em padrões de projetos, defendendo que eles devem ser aplicados acima de qualquer circunstância, às vezes até citando o capítulo do livro que o define. Por favor, não seja essa pessoa.

Neste livro vamos usar *design patterns* para avaliar soluções. Eles serão usados como demonstração de bom ou mau uso de conceitos. Um bom software é aquele que funciona corretamente.

— E até onde devemos ser criteriosos? Devemos sempre procurar a melhor solução?

A resposta para essa pergunta é um pouco complicada, pois qualidade nunca pode vir dissociada de prazo. Devemos sempre produzir software bom o suficiente. Temos que lembrar que nosso trabalho nunca tem fim. Sempre haverá bugs, sempre haverá melhorias e funcionalidades a serem implementadas. Mas sempre há uma data de entrega. Priorize sempre entregar o máximo de funcionalidades corretas dentro do prazo. **Donald Knuth** tem uma máxima que diz **"A otimização precoce é a raiz de todo o mal"** (*"Premature optimization is the root of all evil"*). Invista seu tempo entregando funcionalidades, só otimize quando precisar.

Saber o que é esperado de você

— Qual é o papel do sênior?

Acredita que isso já foi uma polêmica? Polêmica porque muitos nem ao menos sabem qual é o papel de uma pessoa considerada sênior. Ou qual a diferença entre sênior e júnior.

Podemos complicar mais ainda se jogarmos outros termos como arquiteto de software ou engenheiro? Na verdade, o cargo que uma pessoa ocupa é uma combinação de funções e níveis. Podemos encontrar um engenheiro de software back-end sênior, ou uma arquiteta de software back-end júnior. E esses cargos ainda vão depender da empresa em questão. Um arquiteto de software em uma empresa pode ser considerado um arquiteto de sistemas em outra. Há desenvolvedores que respondem como engenheiros. Há engenheiros que respondem como arquitetos. Mas será que conseguimos trazer uma pequena compreensão para esses termos? Para exemplificar isso, vamos criar personas.

PERSONAS

Personas é uma técnica do *Design Thinking* para validar ideias. Sendo bem simplista, consiste em criar personagens, dando nome e características, a fim que você consiga imaginar como elas vão interagir com o sistema.

- João é um engenheiro de 55 anos que não tem muita habilidade com dispositivos digitais
- Maria é uma designer de 27 anos que só usa dispositivos com uma maçã mordida

Dadas as duas personas acima, você deve fazer um sistema que as satisfaça. Como elas se comportarão no seu sistema.

Design Thinking? Ah sim... É um conjunto de métodos e processos que podem ser usados para se pensar sobre um problema. Esses métodos e processos já eram existentes no mundo do Design e foram importados como ferramentas de ideação para diversas áreas.

Para criar as personas, escolhi 3 nomes gerados aleatoriamente e dei cargos a elas.

- **Ana Barbosa Goncalves** é uma Desenvolvedora de Software Júnior
- **Renan Rodrigues Cardoso** é um Engenheiro de Software Pleno
- **Letícia Barbosa Azevedo** é uma Arquiteta de Software Sênior

Essas pessoas têm funções e níveis diferentes. Função significa que os superiores delas esperam que o trabalho feito por uma pessoa seja diferente do de outra. Qual é a responsabilidade de cada uma dentro de um projeto? Nível significa que a maturidade do trabalho feito por elas também é diferente.

A graduação de nível é muito útil e variar de empresa por empresa. Ela diz muito sobre o que é esperado de você dentro do corpo que é uma empresa.

Vamos ver o caso de Ana, ela é uma profissional júnior, o que significa que não se espera muito dela além do mínimo. O mínimo seria conhecer um pouco do desenvolvimento de software e o básico dos conceitos exigidos para vaga. Uma certeza que toda a empresa terá é que ela precisará de ajuda para desempenhar sua função. Ela pode, e deve, sempre pedir ajuda. Pedir para outros a avaliarem e, se possível, ser mentorada por alguém mais experiente que ela.

Já Renan é um profissional um pouco mais experiente. Dentro da sua empresa se sabe que ele pode realizar um bom trabalho sem depender muito da ajuda de outros. Ele pode perguntar também, tanto para os juniores e os seniores, mas pela sua experiência já consegue desempenhar seu papel sem ajuda.

Letícia já é uma profissional sênior, isso significa que muito mais vai ser exigido dela. Ela já terá experiência para mentorar e ajudar muitas pessoas dentro da empresa. É responsabilidade dela, dentro do time, resolver os empecilhos técnicos, mas isso não significa que ela não pode perguntar. Qualquer um pode perguntar a qualquer um e ela deve ser humilde de reconhecer quando não sabe de algo. Ela terá mais experiência, vai saber quando algo pode dar errado, ou quando é preferível não implementar alguma coisa, mas usar algo já pronto no mercado.

— E sobre as funções?

Ana é uma desenvolvedora. É seu papel desenvolver código de qualidade com adequada cobertura de testes, já que não existe código confiável sem testes. Ela deve conhecer sobre engenharia de software, ter a malícia de saber que não há código sem bugs e que o cliente nunca sabe o que quer. Ao meu ver, isso são leis universais. Ana não deve sair *"codando"* confiando no que ela faz e no que o

cliente diz querer. Deve ser criteriosa para levantar questões a fim de resolver possíveis divergência de entendimento.

Para Renan, que é um engenheiro de software, podemos adicionar uma responsabilidade: o desempenho. Apesar de essa função não ser muito comum no Brasil, em multinacionais sempre vemos o papel do *software engineer*. É responsabilidade dele garantir que a solução entregue performance sob determinadas circunstâncias. Ele deverá responder perguntas com números:

- *Essa API suporta um tráfego de 1.500 TPS?*
- *Se o número de usuários dobrar, vamos precisar quadruplicar o número de instâncias?*
- *Essa solução vai escalar facilmente?*

Essa capacidade é construída com uma boa base matemática e algorítmica. É necessário muito estudo para dar respostas que parecem simples, pois a construção desses modelos mentais pode ser difícil.

Por fim, temos a função de Letícia. Mesmo sendo comum, às vezes essa função é pouco entendida. É papel da arquiteta ser responsável pelo desenho técnico da solução. Mas quando se fala desenho, não podemos apenas focar na tecnologia, temos que focar na solução. Todo software só encontra valor quando ele resolve o problema de alguém.

É papel da arquiteta conversar com todas as pessoas envolvidas no projeto e levantar como deve ser uma solução para responder às necessidades dessas pessoas. Esses são chamamos de *stakeholders*, podem ser o gerente, um usuário final, a dona da empresa etc.

— E o líder técnico?

Líder técnico é uma função muito diversa em suas atribuições, que em algumas empresas nem existe. Por isso, para falar dela vou usar a abordagem de Camille Fournier em "A Arte da Gestão". Segundo

ela, o líder técnico pode ou não fazer a gestão das pessoas, mas deve fazer a gestão técnica do projeto.

Gestão, ela define como conversar com outras equipes para esclarecer pontos pendentes, fazer reportes a gerência sobre o andamento do projeto, delegar tarefas para que as atividades sejam concluídas rapidamente e com qualidade ótima. Caso faça a gestão de pessoas, deve dar feedbacks regulares e identificar pontos de melhorias em cada pessoa envolvida no projeto.

Não importa qual é sua função, você tem que responder ao que esperam de você. Coloquei aqui 3 personas para exemplificar o que você deseja ser. Vale lembrar de que você pode ser o que quiser, basta se preparar para isso. E você também precisa responder corretamente ao que esperam de você hoje. Se não sabe, uma dica de como descobrir é perguntar. Perguntar sempre é bom.

Entregar sempre um bom trabalho

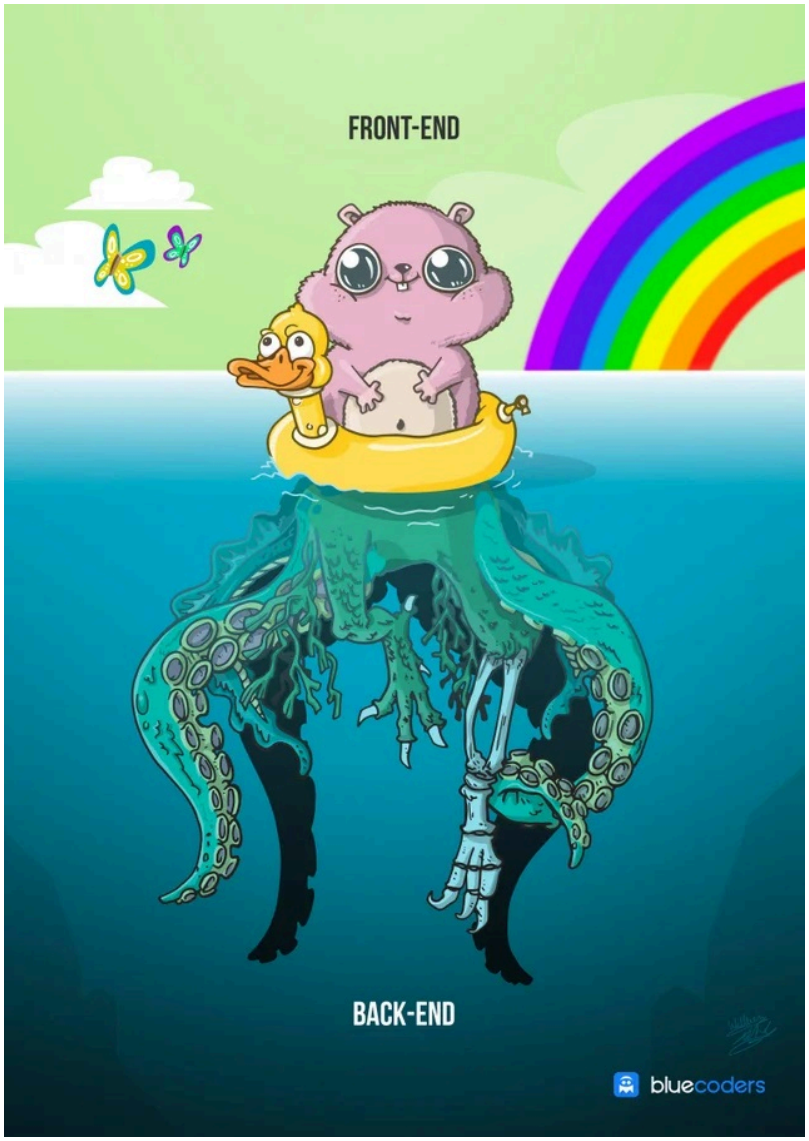


Figura 1.1: Meme ironizando qualidade de software, o Front-end parece bonito e o Back-end é um monstro submarino (https://www.reddit.com/r/ProgrammerHumor/comments/7zfgwg/frontend_vs_backend/)

Talvez você já tenha visto esse meme. Ele foi retirado do canal *ProgrammerHumor* no Reddit. Embora em muitos projetos isso seja verdadeiro, **isso não deveria acontecer**. Qualquer projeto de software pode ter qualidade. Existe ciência e pesquisa nessa área, existem excelentes artigos e técnicas de como melhorar a qualidade de software.

Se você deseja se destacar, precisa sabe criar códigos que são de fácil manutenção. Em *O Programador Pragmático*, David Thomas e Andrew Hunt defendem uma série de posturas que um "programador pragmático" deve ter. Vale lembrar que ser pragmático significa ser objetivo, eficaz. Entre essas posturas está:

- **Dica 5**: Não viva com janelas quebradas
- **Dica 8**: Tornar qualidade uma questão de requisito
- **Dica 14**: Bom design é mais fácil de mudar do que mau design

Essas dicas podem parecer um pouco abstratas, mas são simples de explicar. Na primeira, eles mostram alguns estudos que dizem que uma simples janela quebrada em um bairro pode gerar uma onda de desvalorização de imóveis, isso porque se a janela do seu vizinho está quebrada, você vai cuidar menos da sua casa. Conviver com problemas conhecidos faz você não se preocupar se novos problemas aparecerem. Logo você não deve deixar acumular erros em seu projeto. Faça o levantamento e agende quando essa correção será feita dentro do cronograma do projeto.

As outras duas dicas são mais compreensíveis. Qualidade deve ser um requisito e, se você não a colocar como requisito, isso vai impactar no futuro. Um software com qualidade é um software fácil de dar manutenção e de fácil compreensão. Se os problemas começarem a acumular, um dia a conta vai chegar, seja através de um bug em produção, seja por uma incapacidade de suprir a demanda de requisições.

Ser mestre em ao menos um framework

— Qual é a melhor forma de economizar tempo?

É conhecer o que já está feito e pronto para ser usado. Na minha experiência como desenvolvedor eu já perdi muito tempo reimplementando aquilo que já estava pronto. E também já vi muito código que podia ser removido de um projeto. Isso acontece pela nossa ansiedade em colocar a mão na massa.

Escrever código é bom. Vê-lo funcionando é melhor ainda. Eu amo reescrever alguns frameworks e especificações como prova de conceito. Esse exercício nos ajuda a crescer muito na nossa linguagem, a experimentar conceitos e padrões de projetos. Eu recomendo você a tentar isso no seu tempo livre.

PROVA DE CONCEITO

Já ouviu falar desse termo? Prova de conceito, *proof of concept* ou, simplesmente, *PoC*, é a forma mais eficiente de se validar uma solução. A prova de conceito deve ter dois motivadores básicos, um problema e uma possível solução. Ela não deve gerar um produto, e nem ao menos deve ser usada em produção. Ela serve apenas para validar uma ideia. Seu resultado deve consistir em um miniprojeto funcional e um documento. Tudo que não é relativo à prova de conceito não precisa ser implementado, pois seu escopo é bem limitado.

Sempre que alguém surgir com a pergunta **Será que conseguimos fazer X com Y?** se ninguém tiver uma resposta na língua é o momento de responder **Podemos fazer uma PoC!** Vai lá, cria um pequeno projeto funcional e valide o conceito. Caso não dê certo, não se preocupe, você não jogou tempo fora, você economizou tempo e dinheiro em uma solução que é irrealizável. Caso funcione, *Kudos for you*!

Usar um framework pode ser um pouco limitador para alguns, mas esse é o caminho mais rápido para se entregar funcionalidade. Você não precisa gastar tempo escrevendo um cliente HTTP, ou se preocupando com as novas funcionalidades do HTTP2, basta você usar um framework. Mas também é preciso investir tempo conhecendo o nosso framework a fundo. Muitos dos problemas ou bugs enfrentados no dia a dia podem ser resolvidos apenas com o uso padrão de um framework.

Cada framework tem suas especificidades, seus próprios padrões. Invista um tempo lendo a documentação dele. Sei que essa tarefa é um pouco entediante, mas há muita coisa interessante que está escondida e pode ser decisiva para que você seja considerado um mestre (e, no fim, economizar muito tempo).

Outra atividade que também pode ser interessante é você conhecer o código do seu framework. A maioria deles é *open source*. Dê uma olhada em como as coisas acontecem por baixo dos panos, entenda as engrenagens, veja as limitações, se cadastre nos canais oficiais dele e, se possível, contribua com ele. O mundo da computação é em grande parte construído pela comunidade.

Conhecer os fundamentos da Internet

— E se seu framework deixasse de existir hoje? E se você descobrisse que seu framework não implementa algo muito importante?

Já falamos que precisamos conhecer nosso framework, mas não devemos deixar que ele seja um limitante para nosso conhecimento. Devemos ir além dele. Nos últimos 20 anos vimos uma série de padrões muito respeitados no mercado serem simplesmente jogados no lixo, mesmo resolvendo problemas que não mudaram.

Podemos citar os frameworks dos anos 2000: conheço alguns desenvolvedores que são mestres em JSF, JSP ou Struts, mas estão muito preocupados porque, quando pesquisam, não há uma vaga sequer com essas tecnologias hoje. Isso significa que as placas tectônicas do mundo tecnológico se moveram? Não! A Internet continua sendo feita dos mesmos protocolos, no fundo quase tudo ainda é HTTP, HTML, JavaScript e CSS. O que mudou foi o contexto.

Antes quase todo o desenvolvimento era feito para serviços de poucos acessos. Fazíamos a tela de cadastro de estoque de uma fábrica, ou o sistema que o atendente de telemarketing usava. Mas agora há serviços que são acessados por milhões de pessoas. Agora

é a livraria digital, ou o serviço de entrega de comida. Para responder a isso, houve também o surgimento de alguns novos conceitos, como *Cloud Native*. Vamos explorá-lo mais à frente.

1.2 Quem você quer ser?

Temos um grande desafio à frente. Mas agora abro a pergunta para você. *O que você quer ser no longo prazo?*

Responder a essa pergunta é um grande desafio, mas tendo-a respondida você pode começar a plantar hoje as sementes do que você será daqui a 15 anos.

Neste livro vamos lhe apresentar o que há no mundo back-end. Acredite, ele é muito mais complexo do que parece e ainda temos o grande desafio que está escondido abaixo do iceberg: escalabilidade.

Quem nunca reclamou que um servidor está lento? Em sistemas críticos qualquer otimização é um ganho em escala. Se ganharmos 1ms em cada requisição, isso pode representar algumas horas de servidor, que pode representar menos uso de recursos. Mas vamos deixar isso para depois...

Para começar nossa jornada vamos caminhar um pouco pela história da Internet e apresentar os principais paradigmas arquiteturais.

Capítulo 2
Uma breve história da Internet

— Como chegamos até aqui?

Não posso falar sobre desenvolvimento back-end sem entender o mundo de hoje. E para compreender o mundo de hoje, precisamos conhecer o passado. É através dele que algumas coisas ganham significado. Por isso vamos recapitular a história da Internet. E já que somos desenvolvedores, que tal recontá-la através da nossa ótica?

2.1 O que é a Internet

O mundo era bem mais simples em 1990 e existiam muito menos possibilidades. Se você fosse um desenvolvedor em 1990, era muito provável que você desenvolvesse apenas aplicações Desktop. Mas se você trabalhasse em uma grande empresa, trabalharia em um projeto para *mainframe*, ou seja, seu programa iria rodar em um supercomputador e você o acessaria através de um terminal burro. Essas duas modalidades de desenvolvimento não podem ser consideradas desenvolvimento web, porque ainda não existia a web. Porém, a Internet já existia.

— E como era a Internet nessa época?

Bom, a Internet era basicamente a mesma coisa que é hoje: uma série de computadores interconectados. As placas tectônicas da Internet continuam no mesmo lugar, ela não tem um centro, ela não tem um controle. Seu protocolo principal é chamado TCP/IP, pelo qual duas máquinas conseguem estabelecer uma conexão através de uma técnica chamada de *package switching*.

— Mas como isso funciona?

Para que duas máquinas consigam se comunicar, elas precisam estar conectadas a um *Internet Service Provider*. Este está conectado a outros ISP e fornecerá um endereço para sua máquina, que é o que conhecemos como endereço de IP. Para se conectar com outra máquina, a sua máquina cria um pacote, coloca o endereço origem e destino e envia para todos seus vizinhos, que na verdade é o ISP. Como o ISP não sabe onde está a máquina destino, ele vai enviar para cada vizinho.

Essa operação vai se repetir até que esse pacote chegue ao ISP responsável pela máquina destino, que encaminhará o pacote. Essa descrição é bastante resumida e é óbvio que tem mais inteligência para esse pacote chegar mais rápido ao destino, reduzindo o número de pacotes trafegados na rede. Mas é basicamente assim que funciona o Internet Protocol, vulgo IP.

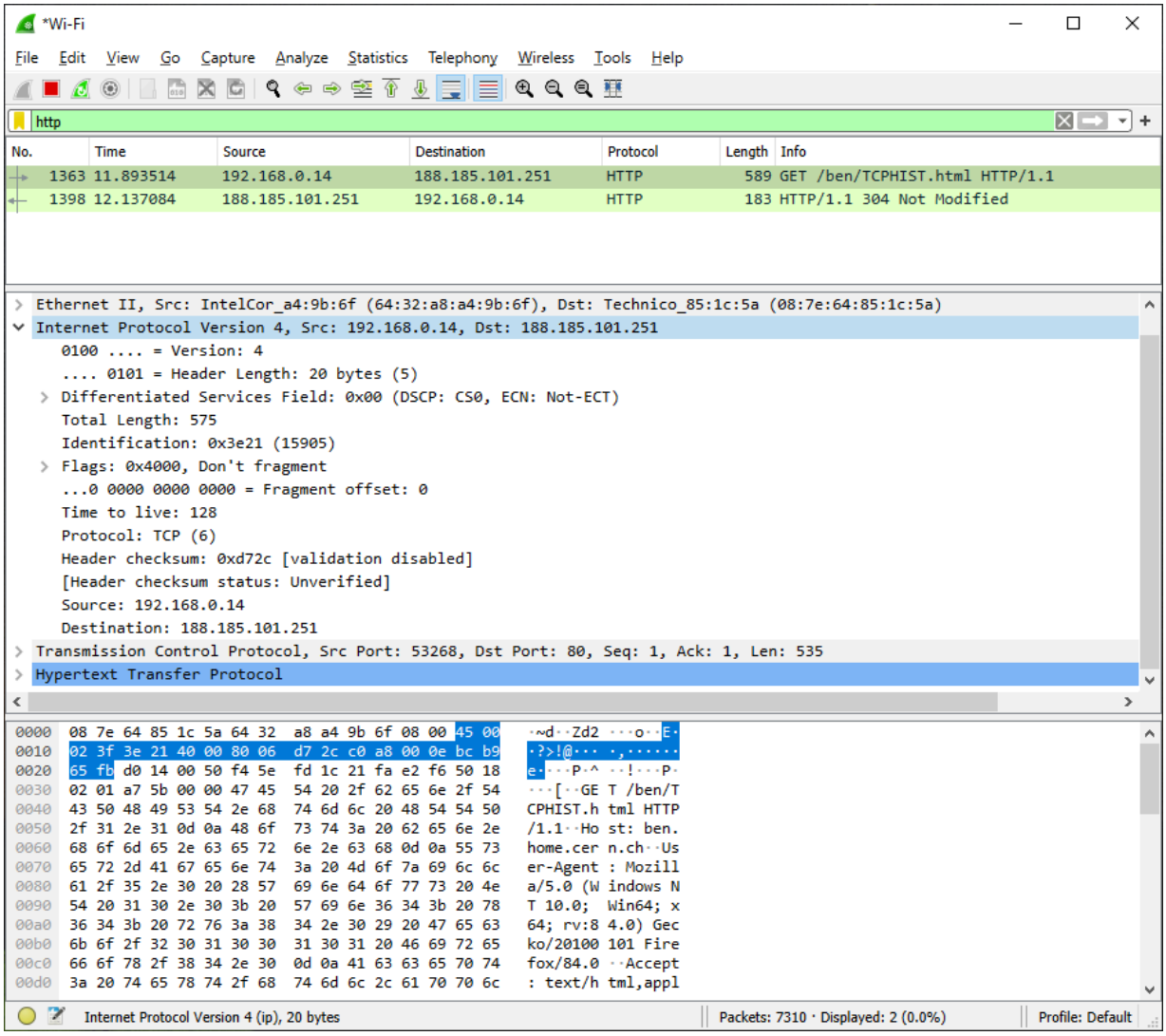


Figura 2.1: Pacote TCP/IP capturado pelo Wireshark. Observe os detalhes do cabeçalho IP

— Então para eu enviar algo pra uma máquina eu preciso saber o IP dela?

Sim e não! O IP pode ser definido diretamente ou traduzido a partir de um nome usando um servidor de DNS. Quando eu procuro `www.google.com`, o sistema operacional que estou usando, antes de enviar uma mensagem, precisa descobrir o endereço físico a que esse nome se refere, isto é, que essa máquina está na rede como `172.217.21.228`.

Nenhum computador consegue enviar uma mensagem para a máquina `www.google.com`, ele não entende essa linguagem. Para isso ele consulta o servidor DNS e envia o pacote para `172.217.21.228`. Isso é o que chamamos de resolução de nome. Todo endereço que usamos na Internet é traduzido em um endereço IP.

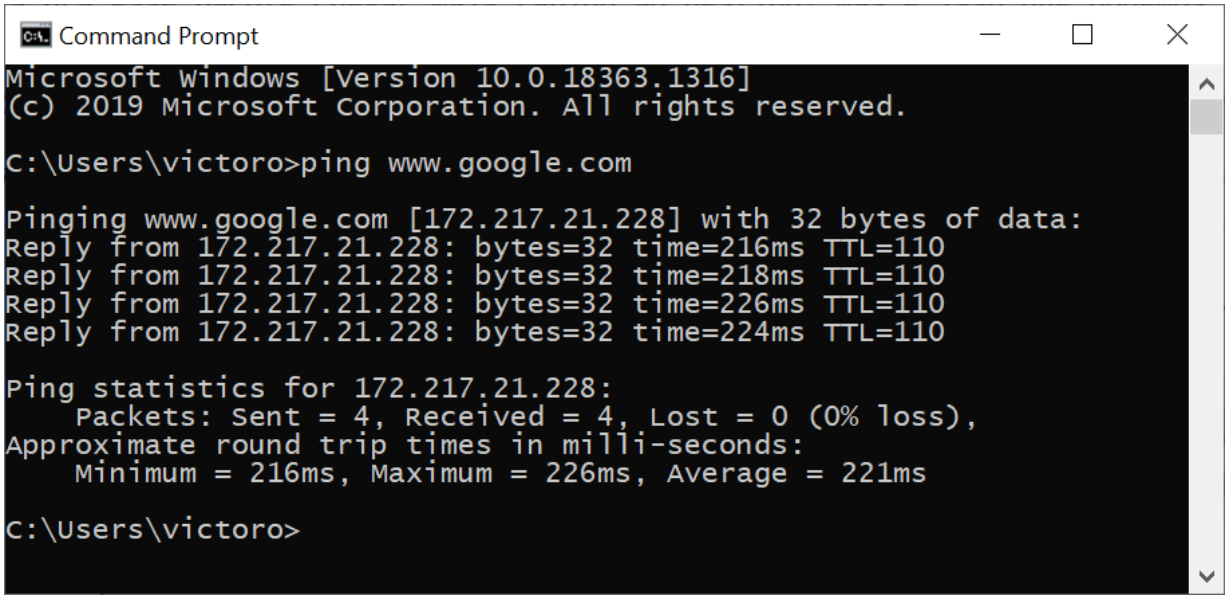


Figura 2.2: Executando um comando ping para descobrir o IP do Google

PACKAGE SWITCHING e **Circuit Switching**

Na verdade, a técnica de **package switcing**, ou comutação de pacotes, é uma evolução da técnica de **circuit switching**, ou comutação de circuitos.

Na comutação de pacotes não criamos uma conexão direta com a máquina destinos, mas compartilhamos várias conexões em comum com várias máquinas. Temos um emaranhado de conexões que se conectam em grandes *backbones*, que são a espinha dorsal da Internet, os cabos que conectam até regiões ou continentes. É a técnica usada pela Internet, de modo que é possível através de apenas uma conexão se comunicar com qualquer máquina na mesma rede. A informação trafega através dessas conexões encontrando o melhor caminho entre as duas pontas da rede.

Já na comutação de circuitos criamos uma conexão direta entre dois pontos. É a técnica das antigas linhas telefônicas. Quando você ligava para um número, uma central telefônica criava um circuito que ligava o seu telefone ao telefone de destino. Se você estivesse fazendo uma ligação, não era possível receber outra ligação, sua linha estava ocupada.

Mesmo não havendo garantia de que um pacote chegue ao outro lado rapidamente, a comutação de pacote é uma grande economia de recursos, pois com a comutação de circuitos há a alocação total de uma conexão para o envio de poucos dados.

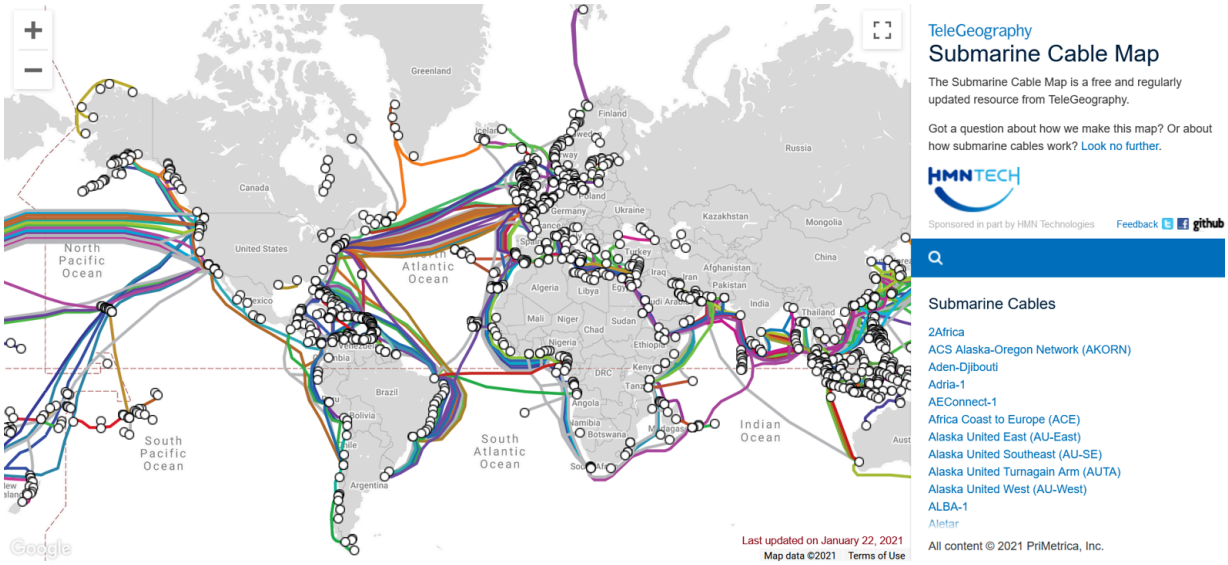


Figura 2.3: O mapa dos cabos submarinos que conectam o mundo. (Fonte: https://www.submarinecablemap.com/ Visitado em 23/01/2021

A Internet é uma tecnologia que surgiu no auge da guerra fria, e um dos requisitos dela era ser completamente descentralizada. Ela não poderia depender inteiramente de uma localidade, pois essa localidade poderia desaparecer do mapa de um dia para outro. Isso não parece factível hoje, mas já viu o seriado **O Homem do Castelo Alto**? Aquele mundo é ficcional, porém o medo de que aquele mundo se tornasse real existia! Era completamente plausível imaginar um país invadindo os EUA e destruindo suas centrais telefônicas. Ou uma bomba caindo em alguma grande cidade e a destruindo completamente. Em todo caso, a Internet não poderia cair, os pacotes precisavam encontrar outro caminho para trafegar entre dois pontos.

Hoje o mundo é outro, e essa tecnologia ganhou outra significância. Podemos nos comunicar sem uma central. O protocolo IP é fruto de uma era, mas ganhou muita relevância em outra. Tudo que precisamos é de um *Internet Service Provider* para acessar qualquer máquina no mundo.

— Mas o que garante que uma mensagem vai chegar corretamente ao seu destino?

Ora, já vimos que o destino é resolvido pelo protocolo IP, mas existe um outro protocolo chamado TCP (*Transmission Control Protocol*). Quando você envia uma requisição para outra máquina, ela pode ser quebrada em vários pacotes menores. A responsabilidade do TCP é garantir que a máquina destino consiga remontar esse pacote completamente. Ele cria uma sequência numérica para cada pacote e, se algum dos pacotes não chegar, o destino sabe exatamente qual está faltando e pode requisitar novamente.

— Mas e se eu não precisar que meus dados cheguem por completo? O TCP envolve retransmissão! Preciso de agilidade!

Há outro protocolo em que não há garantia de chegada de todos os pacotes e nem garantia de ordem. É o UDP (*User Datagram Protocol*), um protocolo que pode ser sua melhor escolha quando você quer transmitir vídeo ou áudio. Os dados são enviados e usados pela máquina destino quando disponível, pois, quando estamos falando de áudio ou vídeo, pequenas falhas são mais aceitáveis que atrasos ou pausas.

Esses dois protocolos se baseiam no IP. Eles não substituem o protocolo IP, mas atuam em camadas. Os pacotes TCP são armazenados em pacotes IP, que por suas vezes são encapsulados em outros pacotes. Isso é o que chamamos Modelo OSI, que contém 7 camadas. No nosso escopo, devemos conhecer as camadas 3 (Rede, implementada pelo IP) e 4 (Transporte, implementada pelo TCP/UDP). As camadas 5 (Sessão) e 6 (Apresentação) não são usadas na Web, sendo elas implementadas acima da camada 7 (Aplicação, HTTP/DNS/etc.). As camadas 1 e 2 garantem que a nossa máquina está conectada a outras e que a conexão é confiável, ou seja, o dado enviado por uma máquina será recebido por outra.

2.2 O WWW e a Internet

— Mas e a diferença entre WWW e a Internet?

Esses são dois conceitos que habitam o mesmo universo, mas são diferentes. Internet nós já definimos anteriormente, é a rede de computadores, mas quando abrimos um navegador entramos na WWW. Quando falamos de Internet, temos que ter ciência do destino das nossas mensagens. Mas é no WWW que podemos navegar, descobrir coisas. Para explicar melhor, vamos agora contar a história do WWW pela visão de quem desenvolve.

As fases do desenvolvimento da Web

Em 1990 começava a história do WWW, Tim Bernes-Lee (https://www.w3.org/People/Berners-Lee/FAQ.html) criou o protocolo HTTP, o HTML e um navegador chamado WorldWideWeb, que já haviam sido propostos em 1989 (http://info.cern.ch/hypertext/WWW/Proposal.html). Seu intuito era compartilhar e conectar conteúdo de institutos de pesquisas, promovendo um protocolo e uma linguagem de apresentação.

Esse foi o pontapé inicial, mas essas conexões foram crescendo exponencialmente, foram adicionados vários centros de pesquisas e empresas. Hoje grande parte da economia mundial depende do WWW. Por isso, vou propor algumas fases desse desenvolvimento e explicar um pouco quais foram as principais tecnologias delas.

- Pré-WWW
- Do WWW à Bolha PontoCom
- Da Bolha PontoCom à Crise do Subprime
- Da Crise do Subprime aos dias de hoje

Pode parecer estranho que as fases da Internet sejam marcadas por duas crises econômicas (Bolha PontoCom e Subprime), mas essas crises alteraram as placas tectônicas da economia mundial, o que possibilitou a mudança de paradigmas tanto tecnológicos quanto de gestão. Elas de fato estão muito relacionadas à Internet e ao modo como os investidores de Wall Street viam as "empresas de Internet".

E ambas as crises estão intimamente relacionadas. Mas para entender isso, é preciso entender alguns conceitos econômicos. O primeiro dele é o **crédito**. Como uma empresa consegue dinheiro? Ela pode ter um investidor ou pode fazer um IPO (oferta pública inicial). Existem pessoas com muito dinheiro que querem fazê-lo se multiplicar, então eles emprestam esse dinheiro para algumas empresas querendo ser donos de parte dela.

Quando falamos de investidores, significa que a pessoa entrou em contato diretamente com os donos da empresa. Já IPOs são feitos através de bolsas de valores, a empresa é avaliada e uma grande porcentagem delas é vendida em pequenos pedaços para pequenos investidores. Chamamos de crédito a relação de confiança entre os investidores e as empresas.

Qualquer investidor vai requerer garantias ao seu investimento. Um investidor individual vai querer participar do conselho da empresa. Já empresas listadas em uma bolsa de valores têm que fazer uma série de relatórios financeiros para provar que têm saúde financeira e estão criando um modelo de negócios sustentável.

Pré-WWW

Nessa época foi criada toda a base tecnológica que é a base da Internet. Foram as pesquisas do Exército Americano, com o intuito de conectar institutos de pesquisas, que criaram o protocolo IP. O protocolo TCP foi criado em 1974 e o UDP em 1980, eles permitem a comunicação de qualquer computador dentro de uma mesma rede.

O e-mail também já existia, mas não era muito difundido. Como ele também é baseado em um protocolo, o SMTP, cada universidade podia ter seu servidor SMTP (*Simple Mail Transfer Protocol*, ou protocolo de transferência de correio simples, em tradução livre) e você poderia enviar email para qualquer pessoa dentro daquele domínio.

Em 1983, foi proposta a *Berkeley Socket API*, com a qual poderíamos facilmente criar programas usando o protocolo TCP. Mais conhecida apenas como *Socket API*, ela consiste em uma série de funções C que encapsulam toda a lógica do protocolo TCP em algumas chamadas relativamente simples. Uma tarefa difícil pode ser traduzida em algumas poucas linhas de código. Esse foi esse o pontapé inicial para o que chamamos de *Socket Server*.

SOCKET SERVER

Socket Server é uma modalidade de aplicação cliente-servidor onde o protocolo de comunicação é exclusivo. Ele só deve ser construído se, e somente se, houver grande necessidade de negócio. Não deve ser confundido em hipótese alguma com *WebSocket*, que será explicado mais à frente.

Sua principal característica é que o protocolo de aplicação é exclusivo, não sendo um padrão de mercado. Deve ser definido pelos desenvolvedores e se possível compartilhado através de bibliotecas.

Do WWW ao estouro Bolha PontoCom

Nos anos 1990, a Internet atingiu o consumidor final, mas ainda não existiam grandes serviços. Essa fase teve um ápice, que foi no dia 10 de março de 2000, quando a realidade cobrou o preço. Como era uma grande novidade, o mercado financeiro viu a Internet como o futuro e por isso resolveu apostar todas as suas fichas nela. Qualquer um, mesmo não entendendo nada de tecnologia, podia ter uma ideia e conseguir muito dinheiro na Bolsa de Valores. Até criaram uma bolsa específica: a Nasdaq.

Nessa época, existia a crença de que, em menos de 5 anos, toda a economia estaria digitalizada. Nem preciso dizer que demorou mais de 20 anos para isso se concretizar, e ainda foi por causa de uma pandemia.

A grande maioria dos serviços existentes eram de notícias e exibição de conteúdo. Essa época é caracterizada por uma grande disponibilidade de ideias e pela ausência de tecnologias para dar suporte a elas. Os serviços que proviam conteúdo eram muitos simples, porque a velocidade da Internet era muito limitada. Vídeos não eram comuns, apenas imagens em baixa resolução. Quem sobreviveu dessa época? Yahoo.com e Amazom.com.

Falando de tecnologia, a maioria dos servidores era construída usando C/C++. Não tínhamos o conceito de Web 2.0 e nem os Processos Ágeis. Isso impossibilitava que essas ideias fossem colocadas em práticas. Desenvolver um código em C/C++ é demorado, não era possível responder a demandas de mercado do cliente final porque os processos de gerência de projetos não permitiam. Imagina você desenvolver um produto usando C em uma estrutura Waterfall (cascata).

Mas foi durante essa época que surgiram as bases que viriam a pavimentar o futuro. Foi nessa época que surgiu o Java e o Java EE.

Java

A primeira dessas tecnologias que surgiram foi o Java. A plataforma Java tem em sua arquitetura uma máquina virtual, a JVM, que cria um nível de abstração onde um artefato compilado em um sistema pode ser executado em outro. Você pode não ver vantagem nisso, mas isso criou uma revolução, pois um sistema desenvolvido em um computador pessoal poderia rodar em um *Mainframe* ou em um servidor web.

Outra vantagem competitiva da linguagem foi a facilidade de desenvolvimento. Java tem uma API muito bem documentada, ela já dava suporte a várias funcionalidades disponíveis no Sistema Operacional que seriam muito complicadas de serem implementadas em C/C++ (*tente criar uma Thread em C*). Outra vantagem é o compartilhamento de código: como Java provê a funcionalidade *Write once, run everywhere* (escreva uma vez, execute em qualquer

lugar) é muito mais fácil criar bibliotecas, não há mais dependência ao Sistema Operacional e nem à arquitetura do processador.

Java EE

O segundo grande avanço também veio do mundo Java, mas não é especificamente a plataforma Java, e sim as especificações Java EE. Para quem não conhece, o Java EE é um paradigma de programação. Em vez de escrever um programa, podemos apenas implementar algumas extensões de uma plataforma. Criamos um pacote que será executado em um Servidor de Aplicação.

Mas qual a grande vantagem que o Java EE trouxe? Primeiro, é criado o conceito de servidor de aplicação. Ao invés de a aplicação conter todo o código necessário para ser executada, existe uma implementação básica de várias funcionalidades que podem ser acessadas por APIs. Isso libera o desenvolvedor da preocupação de implementar funcionalidades básicas, como o próprio protocolo HTTP ou a comunicação com uma base de dados.

Todas as APIs do servidor de aplicação são especificadas, ou seja, podemos ter várias implementações diferentes de servidor de aplicações com a liberdade de poder trocar de implementação, ou mesmo usar várias. Com o Java EE, o desenvolvimento de aplicações ganhou tração, podemos com poucas linhas de código criar um servidor web.

O estado do Java EE hoje

Talvez você tenha lido que o Java EE está morto. Talvez você só encontre vagas ou tutoriais de Spring. Sim, você está certo: Java EE morreu! Longa vida ao **Jakarta EE**!

O que aconteceu foi uma mudança de governança. Quem definia as especificações Java EE eram grupos de trabalhos coordenados pela Oracle. Mas como a política empresarial da Oracle estava dificultando a sua evolução, era mais fácil abrir mão do controle e passar a uma fundação com uma política menos engessada. Em setembro de 2017, todas as especificações foram congeladas e passadas para Eclipse Foundation (sim, a mesma da IDE).

Isso poderia dar agilidade ao Java EE, mas a Oracle não quis abrir mão do nome Java, então foi preciso uma reformulação da marca. O que era Java EE, passou a se chamar Jakarta EE com outra governança. Com mais agilidade, agora poderemos ver uma evolução contínua dessas especificações.

E o Spring? Spring é um framework código aberto da Pivotal. Ele dá suporte a diversas implementações, mas elas nem sempre seguem as especificações Jakarta EE.

Neste livro vamos explorar as especificações Jakarta EE.

Da Bolha PontoCom à Crise do Subprime

Quando a Bolha PontoCom estourou, poucas das grandes empresas sobreviveram. O mercado estava um pouco receoso com "empresas de Internet", então a fonte de capital secou. Era terra arrasada, mas foi nesse território que a Internet amadureceu. Nesse ambiente, havia uma empresa que estava apenas começando, e não tinha conseguido investimento: o Google.

Foram nesses anos que podemos dizer que a maturidade chegou. Foi nesse período que os Processos Ágeis saíram do experimentalismo e se tornaram práticas comuns de mercado. Foi quando a Web 2.0 tomou forma, os sites não eram mais meramente expositivos, mas já havia conteúdo gerado por usuários. Foi ali também que os Servidores de Aplicação foram testados ao máximo até se verem todas as deficiências deles. Foi então que **escalabilidade** se tornou um problema e plataformas distribuídas precisaram ser construídas.

Muitas dessas mudanças não ocorreriam se as empresas estivessem presas ao capital financeiro, pois mudanças de gestão seriam rapidamente rechaçadas pelos investidores. Só assim a cultura ágil e novos modelos de negócios puderam surgir.

A Volta das bases NoSQL

Pode parecer estranho, mas se alguém falasse que usava uma base não relacional por volta dos anos 2000 seria motivo de piada.

Bases relacionais, como a famosa SQL, foram um grande avanço na indústria, tanto que se tornaram padrão de mercado absoluto. Mas o desafio da escalabilidade as colocou em xeque. Valia a pena fazer todas as validações de integridade? Como criar uma chave única em uma base distribuída?

Houve outras forças também, bases SQL são feitas para discos. O que isso significa? A cada leitura em um disco há um custo de movimentação do leitor, e muito das otimizações das bases relacionais são feitas para leitura em disco.

Nesse período se popularizaram outras tecnologias de armazenamento que não são baseadas em disco, e com isso o custo de ler qualquer posição de memória é exatamente igual, pois não há um leitor. Logo, muitas das vantagens técnicas dos bancos SQL não existem mais.

Novas bases surgiram e elas permitiam que fossem construídas de forma distribuída. Não precisávamos mais de apenas uma grande base de dados, mas poderíamos ter várias instâncias da mesma base formando um *cluster*.

CLUSTER

Cluster é um termo bem comum em sistemas distribuídos. É usado quando uma mesma aplicação pode ser executada em várias máquinas distintas, mas elas atuam como se fossem uma só. Podemos dizer que cluster é o conjunto de aplicações que atuam harmoniosamente como se fossem uma. Cada instância da aplicação será chamada de nó.

Quando temos uma aplicação que opera em modo cluster, isso significa que temos mais de uma instância dela rodando em máquinas distintas.

Virtualização

Outra técnica popularizada nesse período foi a *Virtualização*. Talvez você esteja bem familiarizado com o termo *Conteinerização*, e ele na verdade é uma evolução da Virtualização. Apesar de serem tecnologias bem diferentes, foram criadas para resolver o mesmo problema. Elas tentam responder à seguinte questão: Como é possível compartilhar recursos de uma mesma máquina?

Em Virtualização o recurso é dividido em nível de hardware. O Sistema Operacional Hospedeiro vai ceder parte dos seus recursos a outro sistema. Foi um grande avanço, que possibilitou o compartilhamento de recursos e a alocação dinâmica de ambientes.

AWS - Amazon Web Services

No começo dos anos 2000, a Amazon tinha um grande problema. Quando havia picos de vendas, era preciso um uso extensivo de

máquinas, mas essas máquinas ficavam paradas a maior parte do ano.

E se fosse possível alugar esses recursos por um determinado período? Essa foi a oportunidade que levou uma livraria a se tornar a maior fornecedora de infraestrutura cloud. Máquinas virtualizadas podiam ser alocadas sob demanda em um sistema web.

Antes, para você ter um servidor, você tinha que contratar uma máquina, agora você podia reservar uma instância que ficava em um cluster compartilhado. A máquina não existia mais fisicamente, apenas logicamente. Ela se tornara um item a ser adquirido em uma loja.

JavaScript, AJAX, jQuery e a Batalha dos Navegadores

Para fechar, esse é o ponto mais interessante dessa época. É interessante porque felizmente muita coisa ficou no passado. Nessa época a maior dor de qualquer desenvolvedor web era fazer com que sua aplicação suportasse todos os navegadores do mercado. O que facilitava era o domínio do **Internet Explorer** da Microsoft, e o que dificultava era a quantidade de bugs que ele apresentava.

Como a maioria dos serviços desenvolvidos ainda distribuídos internamente nas empresas, tudo podia ser resolvido ao se limitar qual era o navegador. Era comum ter um navegador específico para acessar o site do seu banco, ele normalmente funcionava no Internet Explorer, e você usava o Firefox para acessar blogs e sites de notícias.

Como navegadores era um ponto muito sensível em qualquer projeto de desenvolvimento web, dizia-se que existia uma **Batalha dos Navegadores**, alguns lutando pela padronização e outros lutando para se diferenciar, garantindo sua reserva de mercado.

O JavaScript já existia desde 1995. Sua função inicialmente era alterar o *DOM* (*Document Object Model*). O *DOM*, por sua vez, é a biblioteca que modela os elementos gráficos de uma página HTML

em objetos JavaScript. Ele permitiu programar no navegador, mas ainda havia muitas limitações.

A primeira era a falta de padronização e, para resolver esse problema, foram criadas várias bibliotecas. A mais popular foi o jQuery, que criava uma camada de abstração e encapsulava as chamadas ao navegador. No começo, suas funcionalidades eram apenas cosméticas, sem grande importância. Mas, aos poucos, foi evoluindo, até surgirem verdadeiros legados em JavaScript.

Outra limitação era a criação de um modelo de aplicação cliente-servidor. Até 2005, o JavaScript não suportava chamadas ao servidor. O que isso significava? Todo o código executado poderia apenas alterar a DOM, sem adicionar nenhuma informação. Então foi adicionado o AJAX, *Asynchronous JavaScript and XML* (https://developer.mozilla.org/pt-BR/docs/Web/Guide/AJAX), e com isso finalmente tínhamos uma interface com o servidor.

Mas o que faltava para construirmos verdadeiras aplicações cliente-servidor? Faltava base de código. Ainda tinha muito código a ser escrito para que houvesse uma aplicação que pudesse ser chamada de Front-end e outra, de Back-end. Até esse ponto tudo era uma grande mistura dos dois, o back-end enviava código HTML e JavaScript e esse fazia algumas alterações na página. Não podemos dizer que existia *Client-Side Rendering*, apenas *Server-Side Rendering*.

Dos frameworks que surgiram nessa época, a grande maioria se dedicava a renderizar o conteúdo no back-end. Um dos mais duráveis é o Google Web Toolkit (http://www.gwtproject.org/). Baseado nele foi feito o Vaadin (https://vaadin.com/), que existe até hoje e se tornou uma referência em *Server-Side Rendering*.

Mas alguns projetos insistiram em renderizar no navegador e foram eles os responsáveis pelo sucesso da padronização dos navegadores, pondo fim à batalha. Esses projetos eram verdadeiras bagunças de

código, com inúmeras dependências JavaScript, mas foi por causa deles que surgiram os primeiros frameworks front-end.

Da Crise do Subprime aos dias de hoje

Entre os anos 2000 e 2008, o mercado de capital esteve avesso a "empresas de Internet", os investidores preferiram investir em qualquer outra coisa, mesmo sem ter as garantias de retorno necessária. E foi isso que levou à crise do SubPrime, termo que significa investimento com poucas garantias.

Em um dia de 2008, o mercado financeiro levou um choque de realidade. Haviam passado 10 anos investindo dinheiro em expansão e crédito imobiliário, agora tudo tinha virado uma grande dívida impagável. Logo os engravatados ficaram com muito dinheiro no banco e alguns engenheiros de classe média perderam todo seu dinheiro acumulado por anos (*como em toda crise financeira*). Mas a grande pergunta era: *Onde vamos investir esse dinheiro?*

Alguns ventos já tinham mudado, as empresas de tecnologia não eram vilãs, ao contrário, com o IPO do Google em 2004 e sua charmosa filosofia *"Don't be evil"* (infelizmente abandonada em algum lugar dos anos 2010), as empresas de tecnologia tomaram o status de salvadores da pátria.

Don't be evil

Como as empresas de tecnologias não tinham uma boa imagem, o Google resolve adotar uma filosofia de nunca produzir o mal. Isso levou à crença de que eram empresas com uma nova carga de valores internos.

Várias empresas, que surgiram na mesma época, tinham feito promessas irreais que acabaram gerando a crise da Bolha PontoCom. E a Microsoft, empresa que até então era sinônimo de tecnologia, enfrentava vários processos judiciais por práticas antiéticas. Um desses processos era do próprio governo americano, que acusava a empresa de criar um monopólio de seus produtos dentro do sistema operacional Windows.

Ao adotar essa política, o Google se distanciava desses dois modelos de empresas.

Muitos queriam investir no novo Google, o que abriu espaço para uma série de aberturas de capital. Assim, aquele crédito que estava sem destino específico foi jogado no mercado de *Venture Capital*, indo parar nas mãos de jovens com ideias (boas e péssimas), que constroem empresas ágeis, usando boas técnicas de *Design Thinking*. Tudo que foi maturado desde 1990 estava pronto para ser usado.

O QUE É *VENTURE CAPITAL*?

Venture Capital é um tipo de clube de investimento em capital de risco. Em vez de o fundo procurar poucas empresas para investir, ele vai se aventurar por empresas com estratégias arriscadas.

É um investimento de altíssimo risco, mas que quando tem retorno é superior a qualquer outro tipo de investimento. São os fundos do tipo Venture Capital que criaram o ecossistema de Startups que temos hoje.

Desse período, podemos elencar duas grandes mudanças de mentalidade: o Cloud Native e os Microsserviços.

Cloud Native

Com o advento das primeiras plataformas Cloud, foram necessários novos padrões arquiteturais. Quando esse novo padrão surgiu? Na minha opinião esses novos padrões surgiram com 12 "dicas" para se entregar aplicações JavaScript para plataforma Heroku: os **12 Fatores** (https://12factor.net/pt_br/). Pode até parecer estranho que esse padrão não tenha surgido nas grandes empresas como Amazon ou Google - vale lembrar que a Heroku foi pioneira em *PaaS* (*Platform as a Service*, ou plataforma como um serviço em tradução livre).

Quando falamos de Cloud, podemos ter alguns modelos de contratação:

Tipo	Abreviação	Descrição
On-Premises	On Prem	Se dá quando a empresa resolve criar uma cloud interna,

Tipo	Abreviação	Descrição
		sem contratar nenhum fornecedor.
Infrastructure as a Service	IaaS	Se dá quando o fornecedor de cloud é responsável pela máquina e conexão delas.
Platform as a Service	PaaS	Se dá quando o fornecedor de cloud provê uma plataforma de desenvolvimento que abstrai completamente o conceito de máquina.
Software as a Service	SaaS	Se dá quando uma aplicação é fornecida por completo pela empresa de cloud. As personalizações necessárias são feitas como configuração da aplicação.

Microsserviços

Com a abundância de capital e a pressão por mais retorno financeiro, as grandes empresas começaram a criar grandes serviços, e isso demandava grandes times. Mas logo perceberam que, para administrar um projeto de software, os times tinham que ter um tamanho máximo. A medida do time era de um grupo que comia duas pizzas. Isso mesmo que você leu! Temos uma definição que rodou as grandes escolas de negócios baseada em pizza.

Já que o tamanho máximo de times estava definido, o que houve foi uma explosão do número de reuniões. Cada serviço precisava interagir com muitos outros times e isso gerava complexidade de software. Foi dessa complexidade que surgiram os padrões de microsserviços.

Com padrões de microsserviços é possível se escolher o que escalar. É possível também responder a falhas de serviços. Começou a se colocar em prática toda a ciência de Sistema Distribuídos. Hoje já temos muitos padrões catalogados, alguns dos quais veremos ao longo do livro.

Protocolo HTTP

Nesta parte, vamos conhecer o famoso protocolo HTTP. Ele é a base de todo software construído para internet.

Seu nome completo é **HyperText Transfer Protocol** e é largamente usado por grande parte da população mundial.

Capítulo 3
HTTP, o protocolo da web

Como já falamos, o protocolo HTTP surgiu em 1990, juntamente de uma linguagem de apresentação e um navegador chamado WorldWideWeb. Isso mudou o mundo. Agora existia uma janela em qualquer computador que podia se conectar a qualquer servidor e exibir qualquer conteúdo. O principal fator para ele ser largamente utilizado é ele ser altamente flexível: pode ser usado para exibir conteúdos, compartilhar mídias e comunicar processos independentes. Vamos conhecê-lo mais a fundo?

3.1 Anatomia do protocolo

O protocolo HTTP foi desenvolvido para prover acesso remoto a recursos. Um navegador deve ser capaz de escolher um recurso e executar algumas operações. Para isso, ele define algumas operações: `OPTIONS`, `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `TRACE` e `CONNECT`.

Quase todos usam o HTTP no navegador, mas para você conhecê-lo mais a fundo é preciso ver como ele é de fato usado. Uma boa ferramenta para isso é o **curl** (https://curl.se/) ou **Wget** (https://www.gnu.org/software/wget/).

Usando o curl, você pode fazer uma requisição em `http://www.pudim.com.br/index.html` no modo *verbose* para entender o que está acontecendo.

```
$ curl http://www.pudim.com.br/index.html -v
*   Trying 54.207.20.104:80...
* TCP_NODELAY set
* Connected to www.pudim.com.br (54.207.20.104) port 80 (#0)
> GET /index.html HTTP/1.1
> Host: www.pudim.com.br
> User-Agent: curl/7.68.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Date: Sun, 24 Jan 2021 18:38:10 GMT
< Server: Apache/2.4.34 (Amazon) OpenSSL/1.0.2k-fips PHP/5.5.38
< Last-Modified: Wed, 23 Dec 2015 01:18:20 GMT
< ETag: "353-527867f65e8ad"
< Accept-Ranges: bytes
< Content-Length: 851
< Content-Type: text/html; charset=UTF-8
<
<html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Pudim</title>
    <link rel="stylesheet" href="estilo.css">
</head>
<body>
<div>
    <div class="container">
        <div class="image">
            <img src="pudim.jpg" alt="">
        </div>
        <div class="email">
            <a href="mailto:pudim@pudim.com.br">pudim@pudim.com.br</a>
        </div>
    </div>
```

```
</div>
<script>
    (function(i,s,o,g,r,a,m)
{i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){
                (i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new
Date();a=s.createElement(o),
            m=s.getElementsByTagName(o)
[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m)
    })(window,document,'script','//www.google-
analytics.com/analytics.js','ga');

    ga('create', 'UA-28861757-1', 'auto');
    ga('send', 'pageview');

</script>
</body>
</html>
* Connection #0 to host www.pudim.com.br left intact
```

Observe que primeiro foi necessário resolver o nome `www.pudim.com.br`. Como já falamos, nosso computador não consegue entender isso, então ele usa um servidor DNS para descobrir que se refere a `54.207.20.104`. Resolvido o endereço, ele abre uma conexão TCP na porta 80. Ora, de onde veio essa porta? Todo protocolo tem uma porta padrão que são registradas na *Internet Assigned Numbers Authority* (IANA - https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml).

A requisição

Com a conexão TCP aberta, podemos começamos a usar o HTTP. Observe que estamos usando o HTTP/1.1, em formato texto, isso nos possibilita ler a requisição de 5 linhas. A primeira linha contém a definição do recurso, operação e a versão do protocolo. As três linhas seguintes contêm cabeçalhos e a última linha vazia indica o fim da requisição.

```
GET /index.html HTTP/1.1
Host: www.pudim.com.br
User-Agent: curl/7.68.0
Accept: */*
```

Indo mais a fundo, na primeira linha da requisição temos as principais informações em ordem: o método selecionado, a URI do recurso e a versão do protocolo. A URI foi extraída da URL utilizada, `/index.html` é parte de `http://www.pudim.com.br/index.html`.

Para as linhas do cabeçalho, repare que cada cabeçalho é composto por uma chave e valor. Há alguns cabeçalhos padronizados, onde as chaves e valores são predefinidas assim como o seu uso, mas o protocolo aceita quaisquer valores. Nos próximos capítulos vamos detalhar os valores padronizados desses cabeçalhos, detalhando por caso de uso.

Por fim, a requisição é finalizada por uma linha vazia. Para o protocolo HTTP a quebra de linha é composta pelos caracteres **Carriage Return (13) + Line Feed (10)**, ou seja `\r\n`, seguindo o padrão Windows.

A resposta

Depois de finalizado o envio da requisição, o servidor deve responder com uma resposta. Observe que ela é muito similar à requisição.

```
HTTP/1.1 200 OK
Date: Sun, 24 Jan 2021 18:38:10 GMT
Server: Apache/2.4.34 (Amazon) OpenSSL/1.0.2k-fips PHP/5.5.38
Last-Modified: Wed, 23 Dec 2015 01:18:20 GMT
ETag: "353-527867f65e8ad"
Accept-Ranges: bytes
Content-Length: 851
Content-Type: text/html; charset=UTF-8

[corpo da resposta]
```

Na primeira linha podemos observar que temos: a versão do protocolo usado pelo servidor, o código de status da resposta e sua respectiva frase razão. A frase razão é a mesma informação do código de estado, porém em forma de texto. A informação mais importante aqui é o código de status, ele é quem determina como a mensagem foi processada e é muito importante que seja muito bem escolhido pelo servidor.

Cada status tem um significado específico, mas eles podem ser catalogados em intervalos. Por exemplo, status de 100 a 199 são informacionais, significam que o servidor recebeu a requisição, mas não a processou ainda. Status de 200 a 299 significam que o servidor recebeu a requisição, processou corretamente e a resposta foi positiva para aquilo que estava sendo requerido. Os status de 300 a 309 são retornados quando o servidor recebeu a requisição, mas decidiu não a processar pois o cliente precisa executar alguma operação (consultar um cache, enviar requisição para outro endereço etc.). Já os status 400 a 499 são retornados quando existe um erro na requisição, que pode ter várias razões diferentes como a necessidade de autorização, pagamento ou a simples falta de um parâmetro de busca. E por fim temos os status de 500 a 599, que são respondidos quando existem falhas no servidor. Essas podem ser ocasionadas por erros internos, alta demanda ou qualquer outra eventualidade que não seja dentro do fluxo lógico da operação.

Depois, semelhantemente à requisição, temos alguns cabeçalhos e, por fim, o corpo da mensagem. Observe que, separando a mensagem do corpo temos uma linha em branco, e o tamanho e tipo da mensagem estão especificados na resposta. Podemos afirmar que o protocolo não apenas diz como a mensagem foi respondida, mas qual o seu conteúdo e como ele deve ser interpretado.

A URI

Outro elemento do protocolo que vale a pena ser descrito é a URI. Ela é muito útil quando se fala em construir APIs. URIs são usadas em vários protocolos, você já pode tê-la visto sendo usada para

acessar bases de dados, serviços web, arquivos ou mesmo para baixar um repositório git. Vamos olhar mais de perto e ver por quê?

Uma URI pode ser quebrada em várias partes. Vamos usar a seguinte divisão `Scheme://Authority/Path?Query#Fragment`. Mais à frente vamos detalhar mais algumas dessas partes, mas aqui vamos definir qual a função de cada uma.

Schema (*Scheme*)

O Schema define qual é o protocolo a que aquela URI se refere. Se você acessar o site http://www.google.com.br você saberá que será usado o HTTP sem nenhuma segurança. Já se usarmos https://www.google.com.br, saberá que será usada uma camada de segurança sobre o HTTP. Se você criar um servidor MongoDB e pedir a Connection String dele, verá o seguinte padrão `mongodb+srv://username:password@cluster-name.mongodb.net/test?authSource=admin&replicaSet=nome-da-replica-set&readPreference=primary&appname=MongoDB%20Compass&ssl=true`. Ou seja, o MongoDB utiliza o protocolo MongoDB Wire Protocol (https://docs.mongodb.com/manual/reference/mongodb-wire-protocol/) com registros DNS SRV (https://docs.mongodb.com/manual/reference/connection-string/index.html#dns-seed-list-connection-format), e sabemos isso apenas olhando a URI.

Autoridade (*Authority*)

Autoridade é o trecho da URI que definirá quem será acessado. Podemos extrair daí quais são o servidor, porta, usuário e senha. Na grande maioria das vezes temos apenas o nome ou endereço IP porque os protocolos têm uma porta padrão e não é necessária uma autenticação.

Mas vamos voltar ao exemplo do acesso ao MongoDB? Podemos ver que nossa url está usando o usuário `username` e a senha `password`. A porta pode ser suposta a partir do protocolo. Sabemos que o MongoDB usa a porta 27017 como padrão. Já o IP será resolvido por

um servidor DNS, `cluster-name.mongodb.net` será traduzido para, vamos supor, `10.10.10.10`. Então podemos construir uma URI equivalente alterando esses valores `mongodb+srv://username:password@10.10.10.10:27017/test?authSource=admin&replicaSet=nome-da-replica-set&readPreference=primary&appname=MongoDB%20Compass&ssl=true`. Ambas as URIs são equivalentes apesar de não serem iguais.

Caminho (*Path*)

O caminho vai definir qual recurso será acessado. Você pode observar que o protocolo HTTP tem alguns padrões, e muitas vezes os caminhos `/` e `index.html` são equivalentes, mas isso não está definido no protocolo HTTP. Esse é apenas um padrão entre navegadores e servidores. Caso o recurso `/` não exista, retorna-se o `/index.html`.

Os recursos podem ser definidos como estáticos ou dinâmicos. Estáticos são aqueles onde o servidor não precisa alterar o conteúdo de acordo com os parâmetros da requisição. Já os recursos dinâmicos são gerados de acordo com a requisição.

Parâmetros (*Query*)

Dentro do protocolo HTTP, a *QueryString* faz parte da requisição juntamente com o *Path*. Ela é usada para definir parâmetros de consulta, sendo composta por um conjunto de chave e valor separados pelo caractere `&`. Esses valores são codificados no padrão URL, ou seja, `chave-1=Valor%201&separador=%26&separador-valor=%3D` significa as respectivas tuplas de chave/valor: `chave-1` / `Valor 1`, `separador` / `&` e `separador-valor` / `=`. Você não precisa saber decodificar esses valores, na maioria das vezes seu framework vai fazer isso, mas, caso precise, procure em qualquer site de buscas "url decoder", que existem vários serviços prontos à disposição. Vamos entrar em mais detalhes posteriormente.

Fragmento (*Fragment*)

O fragmento não é usado na requisição HTTP, ele é usado como parâmetro de navegação dentro do navegador. Alterar um fragmento não implicava em uma nova requisição HTTP, por isso, os primeiros frameworks front-end usaram-no como parâmetro para navegação dentro da mesma página. Foi assim que surgiu o conceito da *Single Page Application*. Depois do HTML 5, esse padrão foi abandonado, pois se tornou possível criar um pré-processamento para verificar se a mudança do caminho significa uma nova requisição ou se pode ser processada no front-end.

3.2 Quem define o protocolo?

Uma outra pergunta bem importante é sobre quem define o protocolo. Da mesma forma de outros protocolos, sua definição é resultado de um extenso trabalho em grupos formado por pessoas de várias empresas chaves ou por indivíduos que se dispõem a participar das discussões. Como esses protocolos já estão em uso há bastante tempo, alguns grupos de trabalho não existem mais, porém há outros, como o que podemos ver no rascunho do protocolo HTTP/3 (https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-32).

Toda a documentação já existente pode ser encontrada no site Internet Engineering Task Force (https://tools.ietf.org/). A seguir eu fiz uma lista daqueles que serão usados neste livro:

RFCs	Título	Data
2616	Hypertext Transfer Protocol -- HTTP/1.1	Junho de 1999
2617	HTTP Authentication: Basic and Digest Access Authentication	Junho de 1999
2817	Upgrading to TLS Within HTTP/1.1	Maio de 2000

RFCs	Título	Data
5785	Defining Well-Known Uniform Resource Identifiers (URIs)	Abril de 2010
6265	HTTP State Management Mechanism	Abril de 2011
6266	Use of the Content-Disposition Header Field in the Hypertext Transfer Protocol (HTTP)	Junho de 2011
6585	Additional HTTP Status Codes	Abril de 2012
6749	The OAuth 2.0 Authorization Framework	Outubro de 2012
6750	The OAuth 2.0 Authorization Framework: Bearer Token Usage	Outubro de 2012
7168	The Hyper Text Coffee Pot Control Protocol for Tea Efflux Appliances (HTCPCP-TEA)	Abril de 2014
7230	Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing	Junho de 2014
7231	Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content	Junho de 2014
7232	Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests	Junho de 2014
7233	Hypertext Transfer Protocol (HTTP/1.1): Range Requests	Junho de 2014
7234	Hypertext Transfer Protocol (HTTP/1.1): Caching	Junho de 2014

RFCs	Título	Data
7235	Hypertext Transfer Protocol (HTTP/1.1): Authentication	Junho de 2014
7519	JSON Web Token (JWT)	Maio 2015
7540	Hypertext Transfer Protocol Version 2 (HTTP/2)	Maio de 2015
7797	JSON Web Signature (JWS) Unencoded Payload Option	Fevereiro de 2016
8725	JSON Web Token Best Current Practices	Fevereiro de 2020
8740	Using TLS 1.3 with HTTP/2	Fevereiro de 2020
Draft	Hypertext Transfer Protocol Version 3 (HTTP/3)	Outubro de 2020

Observando na prática

Mas antes de você começar a ler os próximos capítulos, que tal tentar acessar o modo desenvolvedor do seu navegador e ver como alguns serviços usam o protocolo? Para isso basta pressionar F12 e navegar normalmente. Recomendo navegar pela documentação que o Firefox dispõe do protocolo HTTP (https://developer.mozilla.org/pt-BR/docs/Web/HTTP), ela é muito completa e didática.

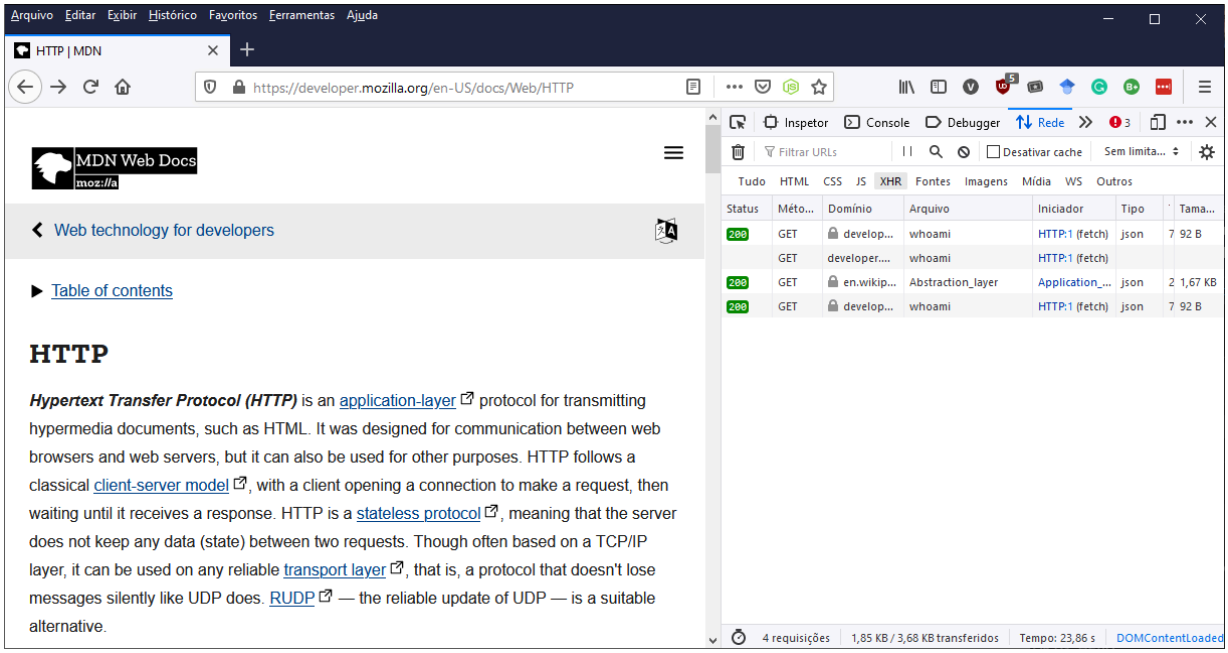


Figura 3.1: Firefox em modo console

Capítulo 4
Discutindo o protocolo

Como vimos, o HTTP é um protocolo que serve para controlar recursos. Agora vamos olhar para alguns casos de uso e ver como eles são endereçados pelo protocolo. Esses casos de uso não são voltados a produtos, mas ao protocolo. Discutiremos como usar o protocolo para criar APIs robustas, explorando todo o potencial dele.

— Mas se o protocolo é aberto, por que discutir como usá-lo? Eu posso usar como quiser!

Essa indagação é completamente válida. O protocolo HTTP é bastante extensível, podemos construir aplicações com ele e explorar uma infinidade de modos de o utilizar. Mas sempre existem padrões e, se seguirmos os padrões, ganhamos em tempo de desenvolvimento e manutenabilidade do nosso sistema. Você vai ver que esses padrões, detalhados nos casos de uso, podem ser utilizados para otimizar sua aplicação.

Esses casos de uso são baseados na especificação do protocolo, a lista não está completa e a pessoa é livre para decidir como usar eles. Em alguns casos, empresas fogem ao padrão para aumentar a segurança ou implementar novas funcionalidades não definidas no protocolo.

4.1 Ferramentas úteis

Para analisar requisições HTTP são necessárias algumas ferramentas. Caso você esteja desenvolvendo uma página Web, usar o console do **Mozilla Firefox** ou **Google Chrome** é essencial. Todo navegador já vem com ferramentas de desenvolvimento que são muito usadas por desenvolvedores front-end, mas com elas também

é possível verificar todas as requisições HTTP, assim como suas respostas. Para acessar, pressione **F12** e navegue para aba **Rede**.

Outra ferramenta interessante é o *Wireshark* (https://www.wireshark.org/). Com ele é possível inspecionar todo o trafego da rede, lendo o conteúdo das requisições, e toda a atividade das interfaces de rede de sua máquina, não somente o protocolo HTTP ou uma porta específica. É útil em casos em que temos vários sistemas rodando no mesmo ambiente e pode carregar *dumps* gerados pelo comando `tcpdump`. Antes de baixá-lo, tenha muito cuidado, verifique se é possível instalá-lo em um computador da sua empresa, pois como ele inspeciona todos os pacotes da rede recebidos pela sua máquina, pode quebrar a privacidade de comunicações, o que pode até resultar em uma demissão por justa causa.

O *Postman* (https://www.postman.com/) é uma ferramenta que possibilita testes de APIs. Com ele podemos fazer requisições HTTP, criar testes automatizados e servidores mocks.

Por fim, existem algumas ferramentas para testes de cargas de APIs. As mais famosas são o *JMeter* (https://jmeter.apache.org/) e *Gatling* (https://gatling.io/). Com eles você pode criar testes automatizados que escalam e produzir estatísticas que serão muito uteis para identificar gargalos de performance. Em ambos os casos, podemos construir testes que vão reproduzir o uso de uma API e podem ser executados inúmeras vezes em paralelo.

4.2 Casos de uso

Para uma análise mais detalhada do protocolo que conhecemos no capítulo anterior, vou propor 11 casos de uso e discutir como cada um é implementado usando HTTP.

1. Decidindo o recurso

2. Decidindo a ação
3. Decidindo o formato
4. Apresentando as credenciais
5. Colocando valores na requisição
6. Adicionando parâmetros de busca
7. Escrevendo a resposta
8. Informando o tipo de resposta
9. Tratando erros
10. Usando o cache
11. Criando um canal *full duplex* de comunicação

1. Decidindo o recurso

Na requisição HTTP, o recurso é identificado pela URI, que vai na primeira linha da requisição. Quando vamos implementar um servidor, a URI pode ser chamada de caminho, *path* ou *endpoint*.

Para vermos um caso real, vamos analisar a API do Twitter. Se selecionarmos as opções *Search Tweets* (https://developer.twitter.com/en/docs/twitter-api/v1/tweets/search/api-reference/get-search-tweets) e *Get Tweet timelines* (https://developer.twitter.com/en/docs/twitter-api/v1/tweets/timelines/api-reference/get-statuses-home_timeline), vamos observar que há uma pequena diferença na URL das duas:

- https://api.twitter.com/1.1/search/tweets.json
- https://api.twitter.com/1.1/statuses/home_timeline.json

Observe que elas têm uma raiz em comum, `https://api.twitter.com/1.1` . O que está acontecendo? O Twitter hospeda todas a suas API sob o DNS `api.twitter.com` e usa HTTPS, que é HTTP sob uma camada de criptografia, como veremos na imagem a seguir.

A URI contém alguns nós, sendo que o primeiro, `1.1` , se refere à versão da API e provavelmente é usado por um servidor que não responderá à requisição, apenas fará o *proxy* dela para outro

servidor contendo a implementação dessa versão. Dados os dois endpoints, provavelmente a mesma implementação do servidor vai receber requisições com URIs `/search/tweets.json` e `/statuses/home_timeline.json`.

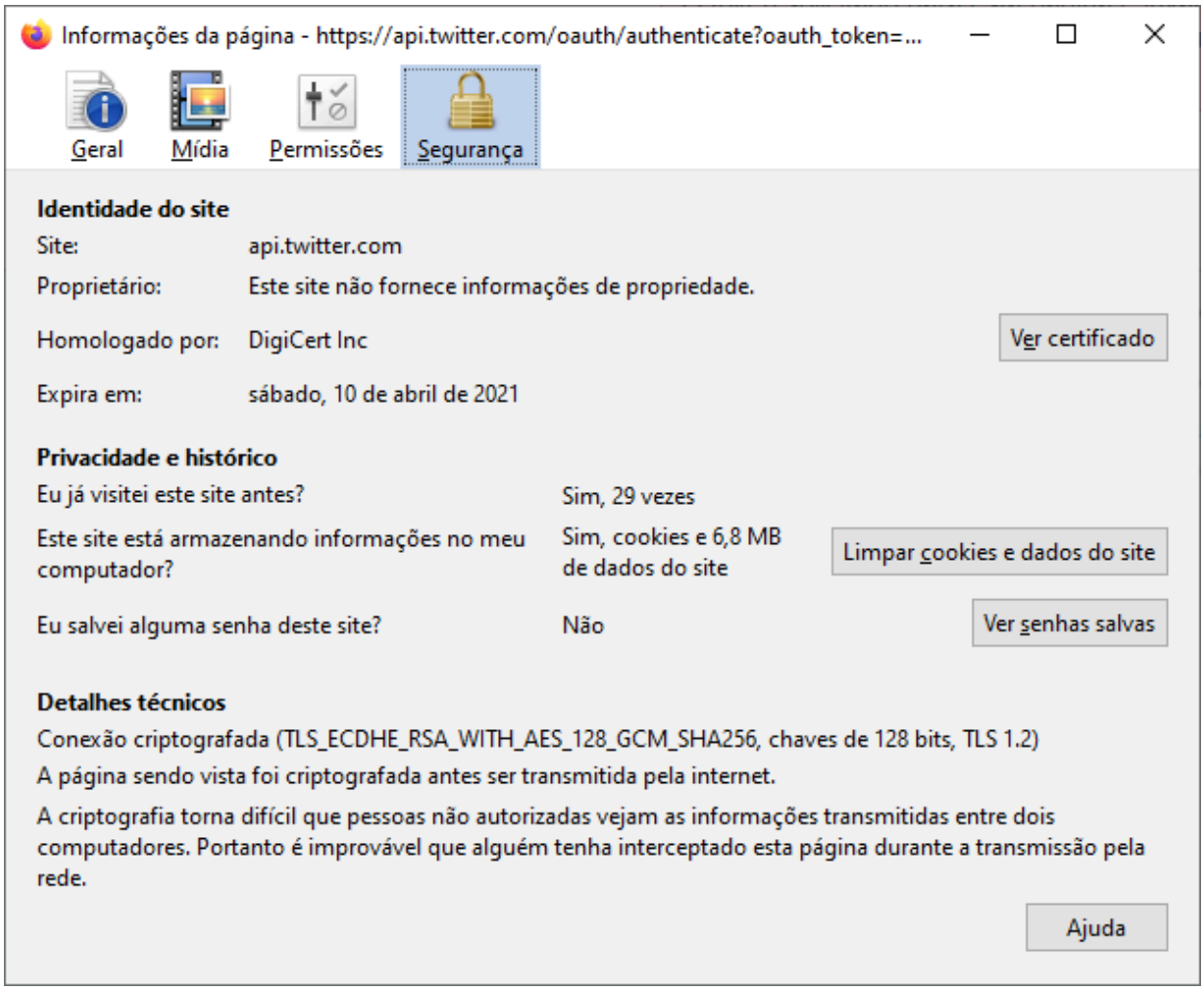


Figura 4.1: Informações de segurança

Podemos verificar qual o endereço físico que resolve `api.twitter.com` usando um simples `ping`. Não podemos afirmar com certeza que é esse servidor que vai responder à nossa requisição; ele pode ser apenas um servidor de proxy ou um API Gateway (tipos de servidores específicos que podem ser configurados usando soluções como o NGINX).

```
$ ping api.twitter.com

Pinging tpop-api.twitter.com [104.244.42.194] with 32 bytes of data:
Reply from 104.244.42.194: bytes=32 time=162ms TTL=54
Reply from 104.244.42.194: bytes=32 time=214ms TTL=54
Reply from 104.244.42.194: bytes=32 time=156ms TTL=54
Reply from 104.244.42.194: bytes=32 time=166ms TTL=54

Ping statistics for 104.244.42.194:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 156ms, Maximum = 214ms, Average = 174ms
```

Muito provavelmente uma máquina com endereço `104.244.42.194` vai receber uma requisição com URL `/1.1/search/tweets.json`. Essa máquina vai copiar essa requisição e enviar para outro processo ou máquina com a URI `/search/tweets.json`.

Proxy Server

Um servidor de proxy significa que é um servidor que não resolve a requisição. Ele pode redirecioná-la para um ou mais servidores, alternando alguns parâmetros.

Podemos criar servidores de proxy facilmente com o NGINX, onde podemos adicionar cabeçalhos ou mesmo alterar o caminho da requisição. Isso possibilita implantar um dos padrões de microsserviços bem conhecido: o API Gateway (https://microservices.io/patterns/apigateway.html). Poderíamos, por exemplo, ter em uma mesma API vários serviços respondendo a diferentes padrões de caminhos. Poderíamos direcionar todas as requisições `/api/produto` para o microsserviço de produtos, e `/api/estoque` para o microsserviço de estoque. No caso do Twitter, é muito provável que as várias versões da API sejam implementadas por serviços diferentes, ficando a cargo de um Gateway de API resolver isso de forma transparente.

2. Decidindo a ação

Dado um recurso, o que fazer com ele? Como isso é expresso na requisição HTTP?

Nós vimos no capítulo anterior que o protocolo HTTP provê acesso a recursos e para isso o método tem um papel essencial dentro do protocolo. O método é o que define a ação dentro da requisição, tanto que ele vem na primeira linha da requisição antes mesmo do recurso.

O uso do método pode variar acordo com o estilo arquitetural escolhido: ele é central para o REST (*Representational State Transfer*), mas ignorado no SOAP (*Simple Object Access Protocol*) e gRPC. Nos próximos capítulos vamos nos aprofundar melhor sobre o que é estilo arquitetural e descrever as características de cada um deles.

Mas já que o método pode ter diferentes significados em diferentes estilos arquiteturais, o que a especificação do protocolo diz sobre cada um? E quais são os métodos existentes?

Vamos então olhar a especificação do HTTP para entender quais são os métodos e como eles são definidos. Na RFC 2616, temos a descrição de cada método, além de uma seção onde se discute se eles devem ser idempotentes.

Idempotência é uma propriedade que a computação herdou da matemática. Quando falamos que uma operação é idempotente significa que, se a aplicarmos uma vez ou várias vezes, o resultado é o mesmo. Então o que significa um método ser idempotente? Significa que eu posso enviar a mesma requisição várias vezes e o efeito deve ser o mesmo do que se apenas uma requisição fosse enviada, considerando que todas as requisições são iguais.

Método	Definição	É idempotente?
OPTIONS	Representa um pedido de informação sobre as opções de comunicação disponíveis na cadeia de pedido / resposta do recurso.	Sim
GET	Significa recuperar qualquer informação (na forma de uma entidade) identificada pelo recurso.	Sim
HEAD	Idêntico ao GET, exceto que o servidor NÃO DEVE retornar um corpo de mensagem na resposta.	Sim
POST	Usado para solicitar que o servidor de origem aceite a entidade incluída na solicitação como um novo subordinado do recurso identificado.	Não
PUT	Solicita que a entidade incluída seja armazenada como o recurso definido na requisição.	Sim
DELETE	Solicita que o servidor de origem exclua o recurso identificado.	Sim
TRACE	Usado para verificar se o recurso está acessível através da requisição.	Sim
CONNECT	Reserva o nome do método CONNECT para uso futuro.	N/A

Para quem já conhece o protocolo HTTP, há alguns detalhes interessantes na tabela. Observe que o PATCH não está listado e que o TRACE e o CONNECT praticamente não são usados. O PATCH foi proposto pela *RFC 2068 Hypertext Transfer Protocol - HTTP/1.1* (https://tools.ietf.org/html/rfc2068), mas, como não havia uma implementação dele, foi removido na atualização. Apesar de os métodos serem definidos em especificações, o uso deles não é livre. Se o framework de desenvolvimento permitir, qualquer método pode ser usado com qualquer finalidade.

Para APIs REST, os métodos ganharam outro significado. Às vezes são chamados de verbos, como podemos ver no *Guideline* (https://github.com/Microsoft/api-guidelines/blob/master/Guidelines.md#74-supported-methods) para APIs REST da Microsoft. Verbos são definidos de acordo com a tabela a seguir:

Método	**Descrição**	**É Idempotente?**
GET	Retorna o valor atual de um objeto.	Sim
PUT	Sobrescreve um objeto, ou cria um objeto nomeado, quando aplicável.	Sim
DELETE	Remove um objeto.	Sim
POST	Cria um novo objeto baseado nos dados fornecidos, ou enviados pelo comando.	Não
HEAD	Retorna os metadados de um objeto para a resposta de GET. Recursos que suportam GET podem suportar também HEAD.	Sim

Método	Descrição	É Idempotente?
PATCH	Aplica uma atualização parcial a um objeto	Não
OPTIONS	Retorna informação sobre a requisição. Permite que um cliente recupere informações sobre um recurso, no mínimo, retornando no cabeçalho identificado por *Allow* os métodos válidos para este recurso.	Sim

Vamos ver mais sobre REST nos próximos capítulos, como esse estilo deu às APIs uma padronização que as torna mais fácil de se compreender.

3. Decidindo o formato

Reparou que até agora não falamos do corpo da mensagem? Tanto a requisição quanto a resposta aceitam corpo. Mas com podemos saber o que há nesse corpo? Na verdade, isso é tanto informado quanto negociado pelo Ciente e Servidor. Essa negociação pode ser feita pelos cabeçalhos `Accept`, `Accept-Language`, `Accept-Encoding` e `Accept-Charset`.

```
Accept: application/json, text/plain, */*
Accept-Language: en,en-US;q=0.8,pt-BR;q=0.5,pt;q=0.3
Accept-Encoding: gzip, deflate
Accept-Charset: iso-8859-5, unicode-1-1;q=0.8
```

Para entender o que a requisição está pedindo, precisamos entender um pouco como funcionam esses campos. Vamos começar com o `Accept-Language`, mas a interpretação do conteúdo vale para todos os outros cabeçalhos, só mudando o escopo.

Na requisição, o navegador envia todas os idiomas, que o usuário configurou, com seus respectivos pesos. No caso, temos que o usuário aceita os idiomas nas ordens de prioridade:

1. Inglês → `en`
2. Inglês Americano → `en-US`
3. Português Brasileiro → `pt-BR`
4. Português → `pt`

Como é que a ordem de prioridade é definida? Através do parâmetro `q=0.3`. Parece estranho porque intuitivamente você entendeu que o separador é o `;`, mas não, o separador é `,`. O `;` está como separador entre valor e parâmetro. Quando omitido, o valor de `q`, *relative quality factor* ou *fator de qualidade relativa* ou simplesmente peso, é `1.0`.

Com esses valores, o servidor vai decidir como será processada a resposta. Se estiver acessando um conteúdo disponível em inglês ou português, vai escolher por português. Se o conteúdo da resposta puder ser exibido como **JSON** ou **XML**, escolherá **JSON**. Essa informação é muito importante para APIs que aceitam internacionalização (*i18n*) e para APIs com várias formas de codificação. Confira o escopo de cada cabeçalho:

Cabeçalho	Funcionalidade
Accept	Especifica os tipos de mídia aceitos como resposta. Podemos escolher **JSON**, **XML**, **YAML** etc.
Accept-Language	Especifica quais idiomas são aceitos como resposta.
Accept-Encoding	Especifica como a reposta pode ser codificada. O cliente pode definir se deseja a resposta em texto plano ou codificada por algum algoritmo de compressão.

Cabeçalho	Funcionalidade
Accept-Charset	Especifica quais charsets são aceitos como resposta.

Ao decidir pelo conteúdo, o servidor deve informar ao cliente quais foram suas escolhas. Isso deve ser feito através do uso dos seguintes cabeçalhos na resposta `Content-Language`, `Content-Type` e `Content-Encoding`. Esses cabeçalhos também devem ser usados pelo cliente em requisições com corpo na mensagem.

4. Apresentando as credenciais

Já que sabemos reconhecer o recurso, o que fazer com ele e como ele será exibido, como podemos implementar segurança nos nossos servidores?

Uma das características do HTTP é que ele é *stateless*. O que isso significa? Na verdade, não há como se guardar o estado da conexão no protocolo. Cada requisição é única e na definição do protocolo não há nenhuma definição de estado, o que quer dizer que para responder uma requisição não são necessárias informações de requisições anteriores. Então como é feita a identificação de usuário e da sessão? Para respondera essa pergunta temos que olhar para os cabeçalhos!

A RFC 2616 define o cabeçalho `Authorization` com essa finalidade: ele deve conter a identificação do usuário. Vale lembrar que, quando se trata de cabeçalhos, o protocolo não determina como as coisas devem ser feitas, apenas dá um direcionamento. Essa RCF não determina como o `Authorization` deve ser usado, citando apenas o termo vago "credenciais".

Mesmo sem ter nenhum padrão definido como obrigatório no protocolo, há um método de autenticação padrão definido na RFC 2617. Conhecida como HTTP Basic Authentication, ela consiste no

uso da tupla usuário:senha codificada em Base64. O cabeçalho ficaria da seguinte forma: `Authorization: Basic Base64(usuário:senha)`. Como a senha é apenas codificada, e não criptografada, esse método de autenticação apresenta algumas falhas de segurança se usada com HTTP. Para resolver esse problema, é aconselhável usar HTTPS, adicionando uma camada de segurança a comunicação.

Pode parecer estranho o uso das palavras *authentication* e *authorization*, elas têm diferentes significados. *Authorization* significa autorização e é quando as permissões de acesso de um determinado usuário são explicitadas. O protocolo HTTP define o cabeçalho *Authorization*, pois lida com recursos, então há o interesse nas permissões. Já o termo *authentication* se refere à autenticação, ou seja, o HTTP Basic Authentication serve para garantir que um usuário é ele mesmo, sua autenticidade, e não suas permissões. Nesse caso, é uma premissa que o servidor saberá transformar usuário e senha em permissões.

A RFC 2617 ainda levanta algumas falhas de segurança em autenticações HTTP. Uma dessas preocupações é referente ao ataque **Man in the Middle**, que acontece quando, entre Alice e Bob, há um interlocutor observando o valor de `Authorization`. Com o conhecimento desse valor, ele pode ser reusado em outras requisições. Quando Alice envia uma mensagem para Bob, o interlocutor pode ler o valor e fingir ser Alice.

Alice e Bob

Quando falamos de criptografia normalmente usamos alguns personagens para representar os casos de uso. Nos papers iniciais de criptografia, quando queriam definir uma comunicação entre máquinas, escolhiam-se letras em ordem alfabética, e se usavam nomes começados por essas letras. Desta forma, os nomes **Alice** e **Bob** se tornaram comuns para descrever a comunicação entre duas pontas. Existem vários outros personagens que você pode procurar na web, na wikipedia temos uma pequena lista (https://pt.wikipedia.org/wiki/Alice_e_Bob). Mas os mais importantes são:

Alice: Alguém que deseja enviar uma mensagem para **Bob** **Bob**: Interlocutor principal de **Alice**, ele não inicia a comunicação, mas pode responder. **Carol** ou **Carlos**: Outros interlocutores secundários, com boas intenções. **Chico**: terceiro interlocutor, mas com más intenções.

Além da Autenticação Basic, há outros modos. Uma forma bem comum é o uso de Cookies, mas eles não são uma boa prática e muitas vezes são usados de maneira antiética por empresas conhecidas como Big Techs. Um Cookie consiste em uma informação adicionada no cliente ou no servidor que será replicada a cada iteração. Definido na RFC 6265, o Cookie é um mecanismo para armazenar tokens, os quais podem servir para armazenar informações do usuário. Uma informação simples como `Cookie: SID=31d4d96e407aad42; lang=en-US` pode conter todo o registro de acesso do usuário em vários endereços da internet. Vale lembrar que o Cookie é replicado apenas por um endereço, mas hoje temos códigos JavaScript de várias empresas rodando nos nossos sites e coletando Cookies.

O Cookie também é usado criar o registro da sessão do usuário, pois ele é uma solução simples onde há um método de expiração do token. Ao se logar em um servidor e este usar o `Set-Cookie: session=e1d8ade7aa00f1a7e66f6f324a5819ad; domain=.mycompany.com; path=/; expires=Fri, 18 Dec 2020 10:37:38 GMT;` , temos um acesso que expirará em 18 de dezembro de 2020. Se o usuário não acessar o servidor novamente até essa data, será necessário um novo token.

Conforme a evolução dos frameworks front-end, um token já não é mais suficiente para o registro da sessão. Ele é muito útil quando temos todas as decisões tomadas apenas de um lado da comunicação. Mas e quando são necessárias informações de acesso dos dois lados? Para isso, a RFC 7519 define o JWT, que é um formato para armazenar informações do usuário. Usando criptografia assimétrica, o token é gerado por uma autoridade, que não necessariamente precisa ser o servidor para o qual está sendo enviada a requisição. Esse pode ser usado no cabeçalho `Authorization` acompanhando o identificador `Bearer` (portador, em tradução livre), definido nas RFC 6749.

Para exemplificar o JWT, vamos imaginar o cabeçalho a seguir, qualquer cliente pode ler esse token e descobrir que ele usa o algoritmo **HS256** e se refere ao usuário **John Doe** com identificador **1234567890**, possuindo as permissões de **ADMIN** e **EDITOR**.

Um token JWT é composto de 3 partes. A primeira, em formato Base64Url, contém um objeto JSON contendo o tipo do token e o algoritmo usado para criptografar a terceira parte do token.

Já a segunda parte, também em formato Base64Url, conterá um objeto JSON com todas as informações do usuário, sendo que alguns campos desse JSON são definidos pela RFC 7919, mas a aplicação pode adicionar o que for necessário.

E a última parte serve para validar a autenticidade, vai conter o conteúdo das duas primeiras partes encriptadas com a chave privada

que não deve ser compartilhada. Cada serviço que contenha a chave pública pode ler essa informação e validar se esse token foi realmente gerado pela autoridade certificadora, ou seja, o nosso serviço de autenticação de usuários.

```
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6I
kpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyLCJyb2xlcyI6WyJBRE1JTiIsIkVESVRPUiJdfQ.
ppRwCyy0PiXgtfRgMZm77CIWHq-Y6sUFtxM2vahlv-k
```

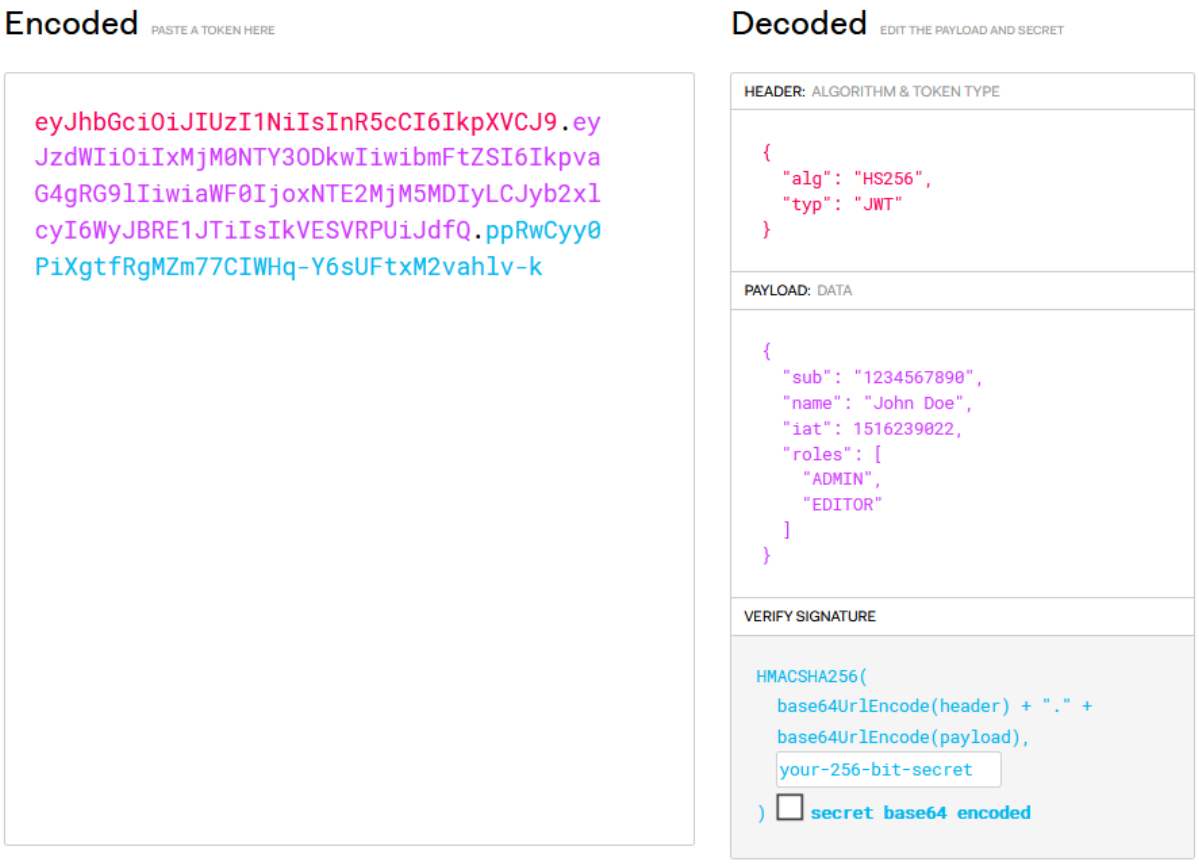


Figura 4.2: JWT decodificado em jwt.io

5. Colocando valores na requisição

Agora que definimos como deve ser o formato do valor de uma requisição, como enviar o conteúdo de uma requisição?

Algumas requisições têm um corpo. A definição do protocolo não se preocupa em definir quais métodos devem ou não ter um corpo associado à mensagem, mas já é um padrão que somente o **POST** e **PUT** tenham um corpo associado.

Isso acarreta em algumas dificuldades para definições de APIs. Imagine um caso em que, para remover um recurso, seja necessário fornecer vários parâmetros, e que esses parâmetros não sejam de buscas, mas façam parte da requisição. Podemos usar **DELETE** com um corpo associado? Sim, é possível, apesar de existir toda uma discussão sobre se essa é a melhor forma resolver esse problema.

Por ser um protocolo bem simples, o HTTP aceita qualquer coisa no corpo da mensagem. Este só é finalizado quando há duas sequências de quebra de linha consecutivas. O protocolo HTTP define a quebra de linha da mesma forma que o sistema Windows, pela sequência de caracteres definida como CRLF (`\r\n`). Mesmo com um marcador fixo, sempre é recomendado o uso do cabeçalho `Content-Length` para evitar que o conteúdo da mensagem seja confundido com a finalização da mensagem.

Caracteres e a quebra de linha

As quebras de linhas são definidas por caracteres de controle. Para entender o que são caracteres de controle, é preciso saber que caracteres não definem apenas letras, mas também outras operações, visuais ou não visuais. Cada caractere é na verdade um número que é entendido por um signo (entenda como sinônimo de símbolo), por exemplo, o alfabeto maiúsculo é definido pelo intervalo de 65 (A) a 90 (Z). Os caracteres de controle (https://pt.wikipedia.org/wiki/Caractere_de_controle) mais usados são o *beep* (7), o *backspace* (8), o *tab* (9), o *line feed* (10) e o *carriage return* (13).

Pare entender melhor a necessidade desses caracteres, é necessário pensar em programas de linha de comando, eles são a maneira mais antiga de formatação de texto. A posição do próximo caractere é chamada de "cursor" e alguns caracteres alteram a posição do curso sem mudar o texto exibido. No caso da quebra de linha, há divergência até hoje de como ela deve ser feita. Em sistemas operacionais da família Unix, o caractere *line feed* significa que o próximo caractere deveria ser posicionado na primeira posição da próxima linha. Já no sistema Windows, para realizar a mesma operação, são necessários dois caracteres: o *line feed*, que define que o próximo caractere vai para a próxima linha, e o *carriage return*, que vai voltar para a primeira posição da linha. Assim dizemos que o Unix usa somente *line feed* (LF) e o Windows usa *carriage return* e *line feed* (CRLF).

Esse valor não é interpretado pelo protocolo, essa função é da aplicação. Assim, não há problema haver discrepância de dados entre o cabeçalho e o corpo da mensagem. Mas vale lembrar que tudo funcionará melhor se o conteúdo vier associado aos cabeçalhos `Content-Type` e `Content-Length`.

```
Content-Type: application/x-www-form-urlencoded
Content-Length: 115
```

O valor de `Content-Type` vai informar a aplicação como o dado no corpo da mensagem deve ser interpretado, ele é um MIME Type (https://tools.ietf.org/html/rfc2045). Um valor pode ser serializado tanto em XML quanto em JSON, mas se esse formato estiver especificado no `Content-Type` a aplicação poderá lê-lo sem nenhum problema.

6. Adicionando parâmetros de busca

E caso tenhamos uma consulta, como podemos adicionar parâmetros sem um corpo na requisição?

Se olharmos bem para o path, veremos que a URI tem uma porção que é comum para vários recursos. Chamamos de *Query String* tudo que vem depois de um `?`. O recurso é identificado pelo caminho que vai do início até o `?`, e o que vem depois é apenas um parâmetro. Analisaremos o caso a seguir. Neste caso, `/a/b/c` é o recurso com dois parâmetros que alteram essa busca.

```
/a/b/c?key1=value1&key2=value2
```

Query strings devem ser usados com muita parcimônia. Esses valores não devem ser usados em requisições que alteram o estado do servidor, apenas em consultas. Isso porque requisições HTTP estão sujeitas a operações de *cache* e o mau uso pode acarretar o não processamento da operação pelo servidor ou da impossibilidade do uso do cache. Veremos como funciona o uso do cache mais à frente.

Um bom exemplo do uso de query strings é na operação de filtragem de recursos e paginação. Vamos supor que temos uma URI `/users` que retorna toda a lista de usuários do sistema. Para retornarmos apenas os usuários com o nome John, podemos adicionar `filter.name=john` e se quisermos apenas a segunda página da busca com 10 elementos podemos adicionar `offset=10` e `limit=10`, assim teremos `/users?limit=10&offset=10&filter.name=john`.

7. Escrevendo a resposta

E como são retornados os conteúdos das respostas? O valor retornado como resposta não é obrigatório. Uma resposta pode vir de outras formas, mas por padrão ela acontece exatamente igual ao corpo da requisição. São usados os mesmos cabeçalhos: `Content-Type` e `Content-Length`. A única diferencia é que o `Content-Type` é decidido pelo servidor de acordo com o valor de `Accept`.

Deve-se ter muito cuidado, pois esse valor pode não existir, dependendo do código do status code da requisição, mas falaremos disso mais adiante.

8. Informando o tipo de resposta

E como saber o tipo da resposta? A requisição foi processada com sucesso? Houve um erro?

Já falamos no capítulo anterior que cada resposta tem o seu respectivo status, é ele quem define o que está sendo retornado. Vamos supor que estamos desenvolvendo uma API para criação de usuários. Para criar um usuário é necessário enviar uma requisição **PUT** para o caminho `/usuario`. No corpo da mensagem devem obrigatoriamente ser definidos todos os dados desse novo usuário.

O que significa se a requisição retornar `200`, `201`, `204`, `400` ou `409`? Cada status pode ser associado a uma resposta completamente diferente. Vamos ver os casos de retornar `200` ou `201`. Essas duas mensagens são válidas e significam que o usuário foi criado com sucesso. Provavelmente, na resposta existe alguma informação que servidor adicionou como o identificador. Já a mensagem `204` significa que o usuário foi criado, mas não existe uma resposta porque não há nenhuma informação importante par o servidor retornar. Já o `400` é retornado quando não há informações suficientes na requisição para se criar um usuário, ou foi identificado um campo faltante, ou o formato é diferente do especificado. E por fim um erro `409` seria retornado se já existe um usuário com alguma

informação existem, por exemplo, se já existir um usuário com o mesmo email utilizado.

É muito importante conhecer cada mensagem, pois muitas vezes um erro pode ser feito ao usar a mensagem errada. Há questões polêmicas como: se uma busca é feita e não se encontra resultado nenhum, deve se retornar `404` ? Na minha opinião não se deve usar, pois a busca foi concluída com sucesso. É mais importante retornar uma lista vazia juntamente com o código `200` . O `404` deve ser retornado se a entidade procurada não é encontrada. Um dos erros comuns no front-end é quando quem desenvolve o back-end não se preocupa em alterar o status de uma resposta, retornando `200` com uma resposta de erro.

9. Tratando erros

O tratamento de erro é a continuação do tópico anterior. Em HTTP os erros são bem definidos e existem código de erros específicos. Imagine o caso de que um parâmetro veio no formato incorreto, ou está faltando um parâmetro obrigatório. Esse caso deve retornar `400` . Uma grande vantagem do HTTP é que os códigos de estado podem ser criados respeitando a separação por tipo. Então se sua API quer criar um novo tipo de erro que seja específico do negócio, é só colocar um código entre 450 e 499.

Retornar uma mensagem de erro não significa que a mensagem não pode ter uma resposta. É uma boa prática em APIs explicar o erro, adicionando um corpo na resposta para demonstrar o que aconteceu e como pode ser resolvido. É preciso muito cuidado para não expor detalhes da implementação como Stack Traces, mas pode ser retornada a informação de quais parâmetros estão fora de padrão, por exemplo.

```
{
    "status": 400,
    "message": "Parameter userId is required!"
}
```

10. Usando o Cache

Já falamos que o protocolo HTTP suporta operações de cache, mas como elas funcionam?

Qualquer operação de cache é complexa tanto na sua definição quando na sua implementação. A primeira questão que precisamos responder é: por que usar cache? Quando temos servidores que são submetidos a uma grande demanda de requisições, qualquer processamento é caro ao servidor, mesmo que seja só para retornar informações. Esses sistemas normalmente distribuem as requisições entre várias instâncias do mesmo servidor e em alguns casos a mesma requisição é feita inúmeras vezes.

Se uma requisição recorrente for respondida da mesma forma sempre, estamos desperdiçando processamento. Poderíamos usar a resposta de uma requisição anterior para responder requisições futuras? A resposta é sim, e esse é o conceito de *cache*. O servidor identifica que o dado não foi alterado e apenas responde ao cliente que o dado anterior pode ser utilizado, evitando processamento, acesso a base de dados e o uso de outros recursos.

O cache de uma requisição pode ser feito por vários estágios, mas para visualizarmos como realmente acontece precisamos saber que em um serviço de alta disponibilidade nem sempre existe só cliente e servidor. Muitas vezes há servidores de balanceamento de carga e servidores de cache em vários níveis. Por fim, temos as bases de dados. Quando mais profundo a requisição chegar, mais será usado da infraestrutura.

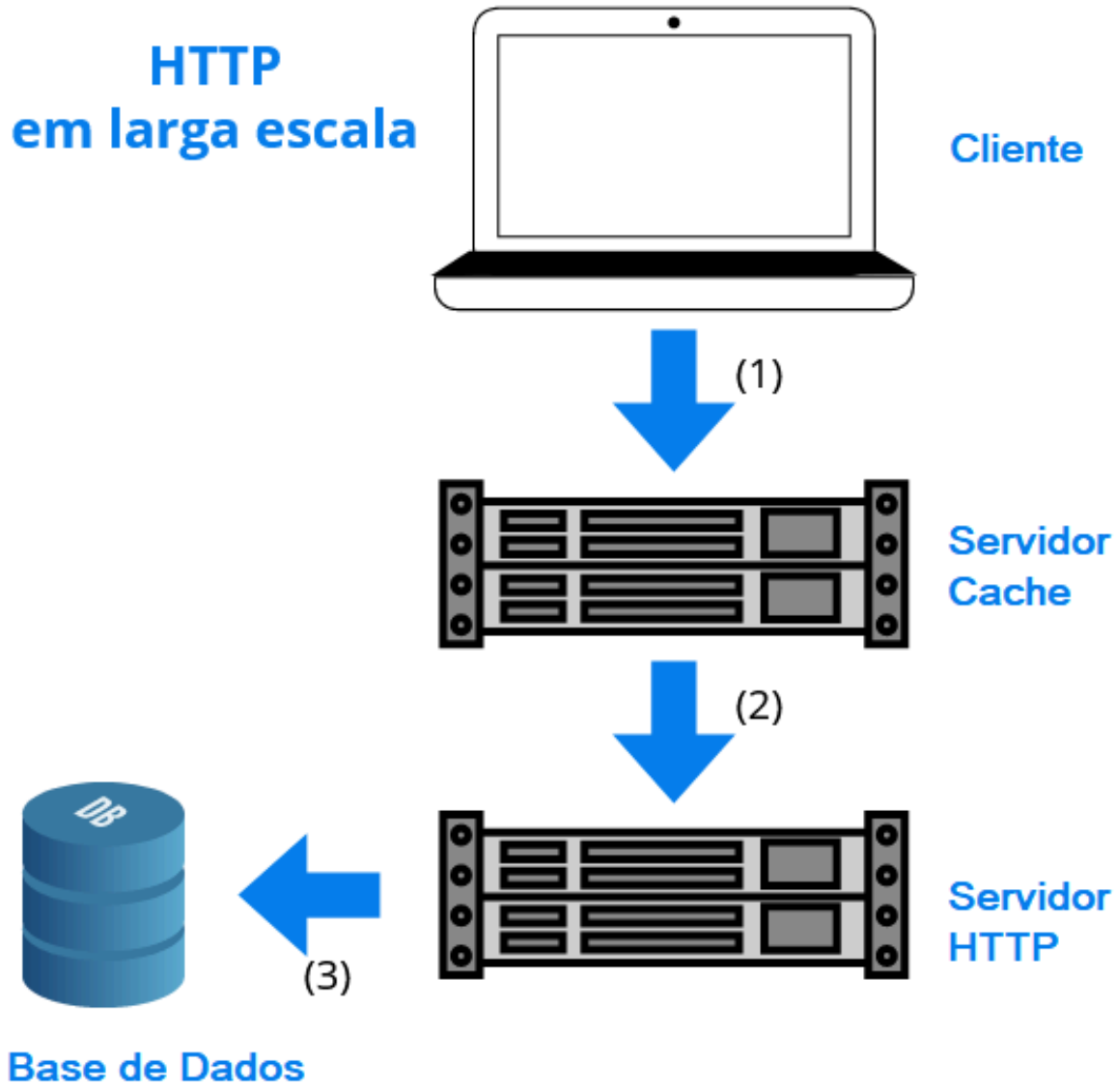


Figura 4.3: HTTP Em larga escala

Podemos classificar os recursos como de dois tipos. Os Recursos Estáticos são aqueles que não podem ser alterados durante o ciclo de vida da aplicação. Estamos falando de arquivos JavaScript ou CSS, imagens. Muitos deles também não precisam ter um nome legível, podendo associar seu nome ao seu conteúdo. Já os Recursos Dinâmicos são alterados durante o ciclo de vida de uma aplicação. O autor de blog pode fazer uma correção no texto, um usuário pode ter seus dados alterados.

Para Recursos Estáticos, a melhor forma de fazer o check é usar o cabeçalho `Cache-Control`. Com ele você pode definir se um recurso vai ter cache ou não, se ele não vai mudar e qual a idade máxima dele. Tem que se ter muito cuidado ao usá-lo se o recurso for mutável. Talvez você já tenha experimentado erros em uma página que nem mesmo uma simples atualização (**F5**) da página resolveu, foi necessário fazer uma atualização com invalidação do cache (***CTRL + F5***) para ela voltar a funcionar. Isso acontecia antes dos frameworks front-end, porque alterações em arquivos não eram refletidas no nome do arquivo. E por que isso não acontece mais? Porque os frameworks inserem no nome do recurso um HASH do seu conteúdo, e muitas vezes o conteúdo de um arquivo JavaScript é gerado a partir de uma série de outros recursos.

A resposta com `Cache-Control` pode ser feita usando o Código de Estado `304 - Not Modified`, ou em alguns casos o cliente nem chega a enviar a requisição. Essa resposta `304` pode ser produzida tanto pelo próprio servidor ou por algum servidor de cache.

Outra forma de implementar cache usando HTTP é usar o cabeçalho `ETag`. Ele representa o estado atual da entidade que não é retornado pelo corpo da mensagem, mas como um cabeçalho da resposta. Se armazenarmos esse cabeçalho, ele pode ser reutilizado em outras requisições tanta para validar concorrência quando cache, para isso precisamos usar os cabeçalhos `If-Match` ou `If-None-Match`, dependendo do caso. `If-Match` é útil quando precisamos validar o controle de concorrência, a requisição só será executada se a entidade não teve o valor de `ETag` alterado. Já o `If-None-Match` deve ser usado para o controle de cache, pois a requisição só será executada quando o valor é diferente. A figura a seguir mostra um exemplo retirado de um caso concreto. Na primeira vez que vai fazer a requisição, o cliente não envia o valor de `If-None-Match`. Mas da segunda, por causa da resposta do servidor, ele é enviado. Na segunda resposta o servidor não respondeu os dados da mensagem, reduzindo o uso da rede.

Controle de Cache

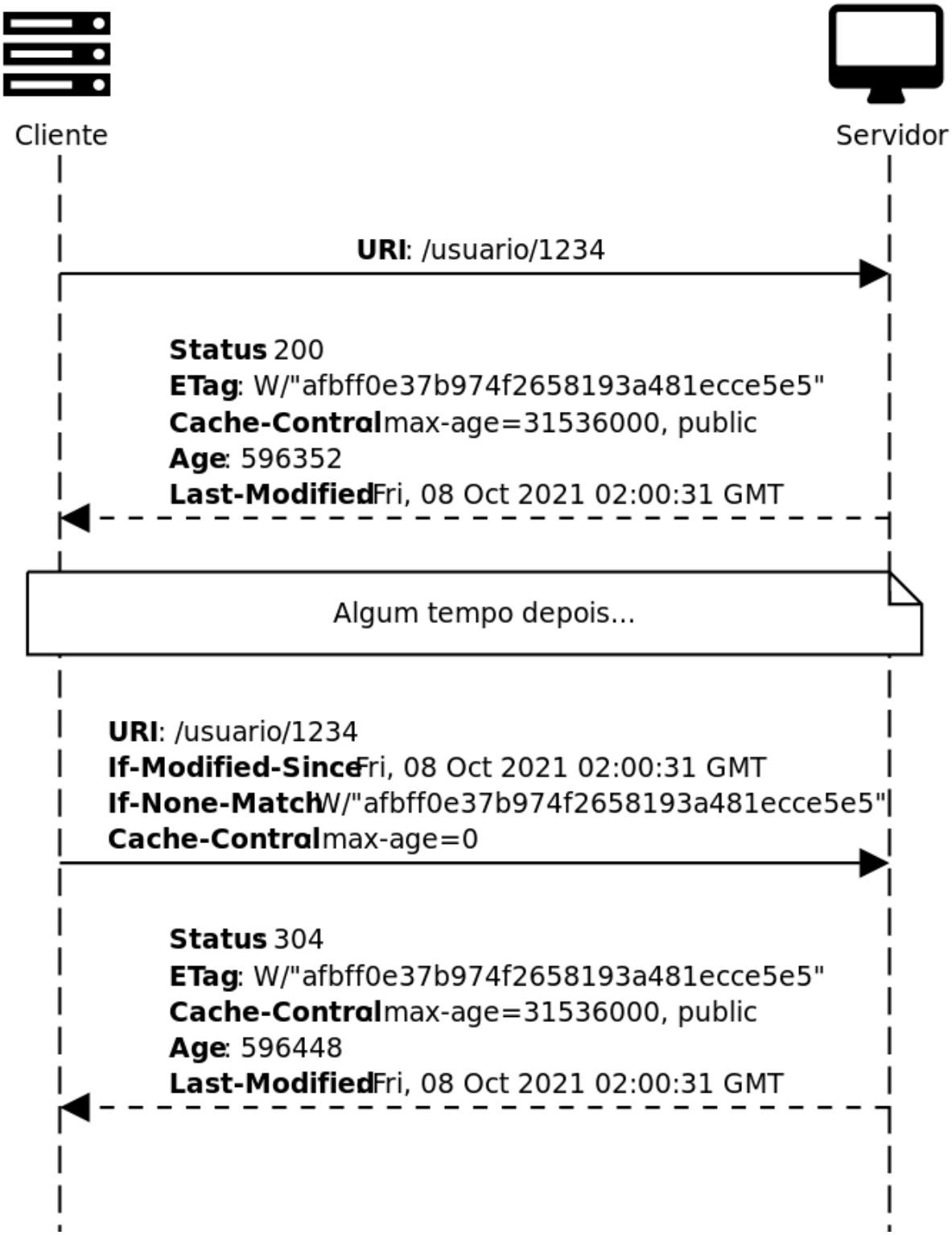


Figura 4.4: Diagrama de sequência

O cache também pode ser implementado baseado na última da de alteração de uma entidade. Semelhantemente ao cabeçalho `ETag`, deve-se usar o cabeçalho `Last-Modified`. Ele deve conter a última data de alteração do recurso, mas os cabeçalhos usados na requisição seguinte devem ser diferentes. Ao invés de se usar `If-Match` ou `If-None-Match`, devemos usar `If-Modified-Since` ou `If-Not-Modified-Since`. No exemplo anterior, vemos que os dois cabeçalhos foram utilizados.

```
Last-Modified: Wed, 21 Oct 2015 07:28:00 GMT
```

Para finalizar, vou apresentar soluções prontas de *caching* que podem ser usadas. Uma delas é o NGINX, que serve tanto como cache como para balanceamento de carga. Caso você tenha contato com soluções complexas de cloud, verá que cada provedor cloud tem uma solução de balanceamento de carga que em muitos casos já oferece opções de cache. Outras soluções são disponibilizadas pelos provedores de Cloud ou pelos gerenciadores de contêineres como Kubernetes e OpenShift.

11. Criando um canal full duplex de comunicação

Uma das falhas iniciais do protocolo HTTP era a ausência de um canal *full duplex* de comunicação. Mas antes de detalhar essa falha precisamos entender o que seria um canal full duplex.

Quando falamos de uma comunicação entre dois pontos, assumimos a existência de um canal. Se Alice envia mensagem para Bob, existe um canal Alice-Bob. Canais podem ser classificados como *half duplex* ou *full duplex* e isso é uma caracteristica do protocolo escolhido. Em um canal *half duplex*, apenas um dos lados pode enviar requisições e ao outro lado cabe apenas responder. Nesse caso, se Alice abre o canal, Bob nunca pode enviar uma requisição, apenas responder às requisições de Alice. Já em um canal *full duplex*, os dois lados podem enviar requisições a qualquer momento.

A princípio, um servidor HTTP só pode responder requisições, não havendo a possibilidade de enviar uma mensagem quando necessário. Uma forma que as aplicações web usaram para resolver esse problema foi enviar sempre uma requisição para verificar atualizações, mas essa "solução" não é performática e não pode ser caracterizada como uma comunicação *full duplex*. Para realmente resolver esse problema foi necessária uma atualização do protocolo.

Por isso, foi criada uma funcionalidade chamada websocket, com a qual é possível enviar e receber mensagens. Um canal websocket é inicializado pelo protocolo HTTP, mas ao ser inicializado a conexão exclusiva é usada para o canal. Apesar de o websocket usar a mesma porta e os mesmos cabeçalhos, ele possui um Schema diferente. Para abrir uma conexão websocket, deve-se usar `ws://www.meuservico.com.br` em vez de `http://meuservico.com.br`. A semântica dessa conexão será muito simular a uma conexão HTTP, mas ela abrirá um canal exclusivo para essa requisição que só se encerará com o fechamento do canal.

A diferença entre o HTTP e o websocket é meramente semântica. Uma conexão websocket é inicializada através do cabeçalho `Connection`, em uma conexão HTTP ele aceita os valores `close` ou `keep-alive`, mas ao usar o valor `Upgrade` a conexão é transformada em websocket alterando a forma como o corpo da mensagem é processado.

Não entraremos no detalhe do protocolo websocket aqui, basta saber que ele deve ser usado quando existe a necessidade de comunicação bidirecional entre ciente e servidor.

4.3 HTTP/2

Falamos bastante do protocolo HTTP/1.1, mas por que não falamos do protocolo HTTP/2?

O HTTP/2 trouxe muitas novas possiblidades, mas ele não é uma reescrita completa do protocolo. Semanticamente, ele continua o mesmo, porém houve um esforço para reduzir a latência na comunicação. Por isso o protocolo deixou de ser em formato texto e se tornou binário, com a possibilidade de multiplexar requisições e de o servidor enviar recursos dos quais o cliente pode necessitar, antes de ser requerido (Push). Com essas alterações foi possível aumentar a eficiência da conexão e diminuir a latência.

4.4 Como se aprofundar?

O protocolo HTTP tem vários outros casos de uso, que podem ser conhecidos lendo as especificações. Mas isso só deve ser feito para quem realmente quer conhecer a fundo o protocolo, é muito raro ser necessário ler (além de muito chato!).

No próximo capítulo vamos detalhar como implementar cada caso de uso discutido usando um framework moderno que oferece uma ótima experiência de desenvolvimento, o Quarkus.

Capítulo 5
Implementando o protocolo com Quarkus

No capítulo anterior mostramos alguns casos de uso para exemplificar o protocolo HTTP. E como criamos uma aplicação web com o protocolo HTTP? A resposta para essa pergunta pode ser direta: é só usar os frameworks de mercado e rapidamente temos um servidor web funcionando. Mas antes é preciso conhecer as funcionalidades do framework que devem ser utilizadas para essa implementação.

Existe um ditado que diz "para quem só sabe usar martelo, todo problema parece um prego". Para que você não caia na tentação de usar a ferramenta errada, vou apresentar as funcionalidades de um dos frameworks de mercado para que você saiba exatamente qual funcionalidade utilizar. Então neste capítulo vamos mostrar como implementar os mesmos casos de uso usando um framework Java.

Caso você queira ver o exemplo completo e funcional pode acessar o código no endereço https://github.com/dev-roadmap/backend-roadmap. Lá você encontrará todo o código e como executar localmente com instruções práticas. Aqui daremos a explicação de cada componente utilizado.

Para construção dessa aplicação foi usado o mais moderno framework Java do momento, o Quarkus.

5.1 O que é o Quarkus?

Quarkus é um framework para desenvolvimento de aplicações *cloud native* que implementa os padrões Jakarta EE. Ao contrário de todos os outros frameworks Jakarta EE, o Quarkus foi desenvolvido com o objetivo de ser portado dentro de contêineres (*containers*). A sua

criação é bastante recente, podendo ser considerado uma reescrita do antigo Thorntail, que por sua vez era uma evolução do Wildfly e do JBoss.

Com a ascensão de novos padrões de desenvolvimento focados em contêineres e microsserviços, os antigos frameworks Java não estavam preparados para resolver as novas demandas. O antigo servidor de aplicação, que continha várias aplicações, ao ser conteinerizado se torna um desperdício de memória.

Servidores de aplicação partem da premissa de que várias aplicações coexistem no mesmo processo. Mas os novos padrões de microsserviços fizeram com que essa prática fosse desencorajada. Um novo conceito surgiu, impulsionado pelos contêineres: a infraestrutura imutável. A arquitetura do Wildfly era otimizada para gerenciar essas várias aplicações, assim como a possibilidade de atualização de uma aplicação sem reiniciar o servidor, mas essas funcionalidades não são mais utilizadas. Foi criado, então, um grupo de trabalho para reescrever o Wildfly de forma que ele se tornasse ideal para desenvolvimento de microsserviços, reduzindo o uso de memória, removendo todas as funcionalidades desnecessárias e otimizando o tempo de inicialização. Como o Wildfly continuaria a existir, esse seria renomeado para Wildfly Swarm, que depois mudou para Thornail para evitar a associação, visto que eles teriam finalidades diferentes.

O QUE É INFRAESTRUTURA IMUTÁVEL?

Para se definir o que é infraestrutura imutável, primeiro é preciso entender o que é infraestrutura mutável. Esse nome "mutável" nunca foi usado para se definir uma infraestrutura, mas esse era um contexto comum.

Para se criar um ambiente de execução era preciso conseguir uma máquina, instalar uma série de dependências e o servidor de aplicação que receberia uma ou mais aplicações. Quando fosse necessário fazer uma atualização, a mesma máquina seria usada e por isso ela precisaria de várias atualizações.

Em situações mais críticas, era necessário atualizar as dependências, o servidor de aplicação e, às vezes, até o sistema operacional. Isso gerava uma série de incertezas: e se essa atualização tivesse um problema crítico? Podemos voltar à versão anterior facilmente?

Depois que surgiram as tecnologias de virtualização, e posteriormente conteinerização, tudo passou a ser mais fácil. Uma máquina não precisa ser atualizada, ela pode permanecer como está. Apenas a máquina virtual, ou o contêiner, é atualizado. > Ao se atualizar para uma nova versão, a infraestrutura antiga é descartada e uma nova é criada. Caso seja preciso desfazer a atualização, a nova versão é descartada e uma nova infraestrutura é criada com a versão anterior. O conceito de infraestrutura imutável surge quando não precisamos mais atualizar máquinas, mas podemos descartar parte máquinas virtuais ou contêineres.

O Thorntail começou então a evoluir rapidamente para um framework voltado a microsserviços e aplicações cloud native, mas ainda havia sérios problemas a serem resolvidos. O mais crítico deles era o tempo de inicialização, pois uma aplicação cloud tem que iniciar muito rápido, visto que novas instâncias dela podem ser

criadas e destruídas automaticamente pelo gerenciador de contêineres. Existiam problemas arquiteturais que impossibilitavam a evolução da base de código antiga, era necessário reescrever grande parte do código para atingir os objetivos desejados.

Essa experiência foi usada para se construir um novo framework baseado nas lições aprendidas do Wildfly Swarm/Thorntail (https://docs.thorntail.io/4.0.0-SNAPSHOT/), citadas na documentação gerada através da prova de conceito feita pelo time do Thorntail:

- **Modularizar artefatos é perigoso**: quando você desconstrói e reempacota os artefatos e dependências de um usuário, muitas coisas podem dar errado.
- **Não substitua o Maven**: deixe o Maven (ou Gradle) lidar com a resolução das dependências. Nós não podemos prever a topologia do repositório de um projeto, ou seus proxies e sua rede.
- **Não se complique ao gerar um pacote**: quanto mais complexo é o nosso pacote final, mais difícil é de gerá-lo utilizando outros sistemas de build que não seja o Maven ou Gradle.
- **Classpaths são complicados**: se diferentes métodos de execução são necessários, você terá problema. Um mesmo código precisa ser executado pelo Maven, ou pela IDE, ou através de um framework de unit-test, ou no ambiente de produção. Para cada ambiente existe uma forma diferente de executar e isso deve ser transparente para o desenvolvedor.
- **Não insista em uberjars**: para contêineres Linux, as pessoas querem ter uma separação clara entre o código da aplicação e o ambiente de execução do código.
- **Testabilidade é importante**: um teste lento é um teste que nunca é executado voluntariamente. Pull-requests demoram para serem validadas. Os usuários gostam de poder testar seu próprio código de forma rápida e iterativa.

- **Facilmente extensível significa ecossistema**: se for muito difícil estender a plataforma, o ecossistema não crescerá. As novas integrações devem ser simples.
- **Contribuições da comunidade são tão importantes quanto o código base**: a detecção automática no WildFly Swarm funcionava apenas com o código base, extensões fornecidas pela comunidade não eram detectadas automaticamente.
- **Certifique-se de que as diferenças entre APIs pública vs. privada sejam claras**: o código interlaçado e Javadocs tornam difícil delinear o que é uma API pública ou uma implementação privada.
- **Permitir componentização**: não queremos decidir todas as implementações, e certamente não as versões, de componentes aleatórios que apoiamos.
- **Seja um framework, não uma plataforma**: frameworks são mais fáceis de integrar em um aplicativo existente; uma plataforma gera muitas restrições.
- **Mantenha testes e documentação**: certifique-se de que a definição de "pronto" inclua testes e documentação.
- **Produtize a complexidade**: quanto maior a divergência entre a comunidade e o produto, mais esforço é necessário para produtizar, complicando qualquer processo para automatizar a produção da comunidade.
- **Complexidade BOM**: BOM significa *Bill Of Materials*, é a forma como o Maven permite gerenciar versões de dependências. Antes do Quarkus, os frameworks usavam vários BOMs, com isso os produtos ficavam instáveis porque alguma dependência não era corretamente atualizada.

Com base nessas lições, o Quarkus começou a ser escrito com alguns objetivos: ser cloud native, ter uma boa experiência de desenvolvimento e ser compatível com especificações Jakarta EE.

5.2 Jakarta EE e JAX-RS

No capítulo 2 mostramos que o Jakarta EE é um conjunto de especificações que facilitam o desenvolvimento web. Dentre elas, vamos focar no JAX-RS (https://eclipse-ee4j.github.io/jaxrs-api/), que é a especificação pensada para facilitar a implementação de APIs REST. Vamos ver no próximo capítulo o que significa uma API REST, por enquanto vamos focar nos detalhes do protocolo HTTP.

O JAX-RS não é a única especificação que implementa o protocolo HTTP, o Jakarta EE também define a Servlet API (https://javaee.github.io/servlet-spec/downloads/servlet-4.0/servlet-4_0_FINAL.pdf). A Servlet API é muito mais antiga que a JAX-RS, possuindo estilos de desenvolvimento que hoje podem até parecer arcaicos, por isso vamos focar na JAX-RS.

Para criar uma API usando JAX-RS, precisamos conhecer um conjunto de anotações definidas na especificação. Toda configuração é feita exclusivamente no código Java, sem a necessidade de arquivos XML. O que vamos apresentar aqui não é uma demonstração completa da especificação, mas sim formas como ela ser usada para implementar os casos de uso do protocolo HTTP que discutimos no capítulo anterior. Os objetivos principais da especificação JAX-RS são:

- **Ser baseado em POJOs**: a implementação é projetada de forma a ser fácil expor POJOs como recursos da API.
- **Ser centrado em HTTP**: se estamos falando de REST, estamos falando de HTTP, e o centro da API é o protocolo. Você terá contado com URIs e verbos e deve entender o que é isso.
- **Ser independente de formato**: uma API REST pode responder XML, JSON ou qualquer outro formato. Você precisa mudar seu código? Pois é, JAX-RS deixa seu código independente.
- **Ser independente de contêiner**: essa é uma feature das especificações Jakarta EE. Você pode escolher a implementação

que seu código vai rodar, não ficando atrelado a um framework/empresa.

> **POJO**
>
> POJO é um acrônimo para *Plain Old Java Object*, que significa velho e simples objeto Java. É um termo usado para referenciar um padrão de classes Java que não dependem da herança de interfaces ou classes de frameworks externos.

— Mas por que usar anotações? Não seria mais fácil definir tudo isso em um arquivo? Ou configurarmos chamando uma biblioteca?

Anotações são bastante úteis porque isolam lógicas que não pertencem ao modelo de negócios. Um dos paradigmas das especificações Jakarta EE é que todo código que não pertencer ao modelo de negócios deve ser adicionado como um adereço. Esse paradigma facilita o desenvolvimento por reduzir a quantidade de código desnecessário, já que algumas configurações passam a ser feitas fora do código e são implementadas pelo framework.

5.3 Casos de uso

Agora vamos retomar os casos de usos do capítulo anterior de um outro ponto de vista. Vamos tentar demonstrar qual é a melhor forma de implementar cada caso de uso.

Para exemplificar usando Quarkus, vamos rever todos os casos de uso do capítulo anterior com um sistema em mente para que nossa demonstração não seja tão abstrata e para sermos mais objetivos. Nosso produto será um gerenciador de requisições de mudança, uma alternativa ao Jira ou Trello, por isso vamos usar os recursos Usuário e Tíquetes.

1. Decidindo o recurso

Ao se pensar uma API, a primeira atividade que precisamos fazer é definir os recursos que ela vai expor. Toda API HTTP é baseada em recursos, em nosso caso podemos definir que teremos dois recursos básicos: Usuários e Tíquetes. Usuários podem criar tíquetes e tíquetes podem ser atribuídos a usuários.

Para cada recurso vamos criar uma classe Endpoint, que será a interface do nosso sistema que responderá a requisições HTTP. Assim precisaremos criar as classes `UsuarioEndpoint` e `TiquetEndpoint`. Cada uma dessas classes terá uma operação básica que retornará todas as entidades cadastradas como recurso. Mas, como tíquetes podem ser associados a usuários, podemos criar um novo endpoint que será dependente do usuário para listar todos os tíquetes associados a determinado usuário.

A classe `UsuarioEndpoint` possui todas as informações de que precisamos para entender como podemos explorar ao máximo os recursos do JAX-RS. Observe que ela tem uma série de anotações que permitem ao Quarkus identificar onde e como pode usar as funcionalidades implementadas. As anotações `ApplicationScoped` e `Inject` não são referentes ao JAX-RS, mas ao Java CDI, uma outra especificação muito importante no Quarkus.

O Java CDI é responsável pelo controle do ciclo de vida dos objetos dentro de uma aplicação. Tudo que precisamos saber é que `ApplicationScoped` significa que essa classe será usada para se criar apenas um objeto com esse tipo nessa aplicação. Ela é usada em contraste com `RequestScoped`, `SessionScoped` e `ConversationScoped`. Já a anotação `Inject` é usada para se adquirir a instância de objetos gerenciados pelo CDI.

```
import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
```

```
import javax.ws.rs.PathParam;

@Path("/usuario")
@ApplicationScoped
public class UsuarioEndpoint {

    @GET
    public List<UsuarioResponse> listar() {
        // implementação
    }

    @GET
    @Path("/{id}")
    public UsuarioResponse encontrarPorId(@PathParam("id") Long id) {
        // implementação
    }

    @GET
    @Path("/username/{username}")
    public UsuarioResponse encontrarPorUsername(@PathParam("username")
String username) {
        // implementação
    }

    @Inject
    TiqueteUsuarioEndpoint tiqueteUsuario;

    @Path("{usuarioId}/tiquetes")
    public TiqueteUsuarioEndpoint tiquetesPorUsuario() {
        return tiqueteUsuario;
    }
}
```

As anotações referentes ao JAX-RS são a `Path`, `PathParam` e a `GET`. Por enquanto, vamos focar nas `javax.ws.rs.Path` e `javax.ws.rs.PathParam`. `Path` é usada para definir qual é a URI do recurso, que é o que ela aceita como parâmetro. Essa anotação pode ser usada na classe e em métodos. Quando é usada na classe, define uma URI básica que pode ser estendida em métodos.

No nosso exemplo, definimos uma classe que responderá a todas as requisições começadas por `/usuario`. Também definimos alguns métodos que vão responder a padrões específicos como `/usuario/{id}/`, onde `id` é um número que identifica o usuário, e `/usuario/username/{username}`, onde `username` é um nome de usuário. Esses são valores que são variáveis na requisição, uma requisição `/usuario/1234` será respondida pelo método associado ao caminho `/usuario/{id}/` e `/usuario/username/admin` será respondido pelo método associado a `/usuario/username/{username}`.

Para termos acesso ao valor dessas variáveis, precisamos usar a anotação `PathParam`. Ela depende da anotação `Path` e o seu valor deve estar contido na URI definida em `Path`. `PathParam` pode ser definida tanto para parâmetros de um método quanto para campos da classe em questão.

Todos os métodos retornam o POJO usuário, ou uma lista de usuários, exceto o método `tiquetesPorUsuario`, que retorna uma classe do tipo endpoint. Isso deve acontecer quando desejamos criar endpoints dependentes de recursos. Observe que nesse caso não estamos retornando a classe `TiqueteEndpoint`, mas `TiqueteUsuarioEndpoint`. Essa outra classe se difere basicamente por não precisar ser marcada com a anotação `Path`, pois ela já está definida no método. E todos os métodos dessa outra classe podem referenciar o parâmetro `usuarioId` usando `PathParam`.

2. Decidindo a ação

Dado que os recursos já estão definidos nos casos de uso anteriores, como podemos adicionar ações neles? Você pode ter reparado que não descrevemos o uso da anotação `GET`. Essa anotação de define o verbo HTTP que estamos usando. No capítulo anterior usamos a expressão método HTTP, mas neste capítulo vamos preferir a expressão verbo HTTP para evitar confusão entre método HTTP e método Java. Os verbos HTTP são definidos como métodos nas especificações, mas ambos os termos se referem à mesma entidade. Daqui em diante, sempre que nos referirmos a verbo, estamos

falando sobre o protocolo HTTP, e sempre que nos referirmos a método estamos falando sobre a implementação em Java.

Para os verbos, o JAX-RS define as seguintes anotações: `GET`, `HEAD`, `DELETE`, `OPTIONS`, `PATCH`, `POST` e `PUT`, que devem ser usadas para marcar o tipo de requisição que método deve responder. O JAX-RS também define a anotação `HttpMethod` para que seja possível estender o protocolo implementando outros verbos.

Para que possamos demonstrar como usar os verbos, vamos adicionar duas novas operações no endpoint do tópico anterior. Vamos habilitar a criação de um novo usuário e a edição de um usuário de duas formas diferentes.

```
@Path("/usuario")
@ApplicationScoped
public class UsuarioEndpoint {

    // [Código omitido]

    @PUT
    public UsuarioResponse novo(CriarUsuarioRequest request) {
        // implementação
    }

    @POST
    @Path("{id}")
    public UsuarioResponse editar(@PathParam("id") Long id,
EditarUsuarioRequest request) {
        // implementação
    }

    @PATCH
    @Path("{id}")
    public UsuarioResponse aplicarEdicao(@PathParam("id") Long id,
PatchUsuarioRequest request) {
        // implementação
    }
}
```

Podemos observar nessa classe que definimos mais de um método com o mesmo caminho. O método `novo` será acessado usando o mesmo recurso do método omitido `listar`, a diferença entre os dois será o método HTTP e o corpo da mensagem.

Como já discutimos, alguns métodos HTTP usam corpo e este é definido por um simples POJO sem nenhuma anotação.

Os métodos `editar` e `aplicarEdicao` também usam o mesmo recurso, mas têm funcionalidades completamente distintas. Normalmente, o `POST` é usado quando todos os campos devem ser alterados. Depois da execução todos os valores do usuário na base de dados serão equivalentes àqueles em `EditarUsuarioRequest`. Já o `PATCH` é usado para alterar apenas alguns campos. Campos sem valores não devem ser alterados.

3. Decidindo o formato

O próximo passo é definir quais formatos podemos expor na nossa API. Como estamos criando uma API back-end, vamos começar pelo uso do cabeçalho `Accept` e sua respectiva resposta, o `Content-Type`.

A especificação JAX-RS define duas anotações que devem ser sempre utilizadas `Consumes` e `Produces`. A decisão de qual formato utilizar não deve ser tomada pela pessoa desenvolvedora, mas pelo Quarkus, afinal isso não faz parte da lógica de negócios. Quem desenvolve deve apenas especificar o POJO a ser retornado e quais os possíveis formatos em que aquele POJO será enviado. Em muitos casos um mesmo código pode ser utilizado para fornecer dados de dois formatos diferentes. Mas em outros casos é preciso algumas configurações um pouco mais específicas, como veremos no caso do XML.

A seguir vamos redefinir a classe `UsuarioEndpoint` para observarmos o que devemos alterar para permitir que a API use tanto JSON quanto XML.

```
@Path("/usuario")
@ApplicationScoped
public class UsuarioEndpoint {
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<UsuarioResponse> listar() {
        // implementação
    }

    @GET
    @Produces(MediaType.APPLICATION_XML)
    public UsuariosResponse listarXml() {
        // implementação
    }

    @PUT
    @Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
    @Consumes({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
    public UsuarioResponse novo(CriarUsuarioRequest request) {
        // implementação
    }
}
```

No método `novo` foi usado mais de um valor para cada anotação. Repare que `Consumes` e `Produces` aceita tanto um valor único quanto uma lista de valores. Ao definir o formato JSON, será necessário apenas adicionar a dependência `quarkus-resteasy-jsonb` do Quarkus ao projeto. Mas ao definir o formato XML é preciso adicionar, além da dependência `quarkus-resteasy-jaxb` , algumas anotações aos POJOs retornados. A classe `UsuarioResponse` deve ser definida com anotações da especificação JAXB.

```
@XmlRootElement(name = "usuario")
@XmlAccessorType(XmlAccessType.FIELD)
public class UsuarioResponse {
    // implementação
}
```

JAXB E JSON-B

Tudo no mundo Jakarta EE é feito através de especificações, e não será diferente para a serialização de objetos. As duas especificações mais conhecidas, e úteis, são a JAXB (*Jakarta XML Binding*) e a JSON-B (*Java API for JSON Binding*). Elas vão definir tanto como ler e escrever um objeto através da sua própria API e como preparar a classe para permitir o uso da API.

A diferença entre essas especificações e as bibliotecas de serialização XML/JSON é que o processo de serialização será transparente. Quando se usam essas especificações, não é necessário se preocupar com detalhes de implementação do formato, mas ao usar a biblioteca diretamente é preciso conhecer esses detalhes.

Essas anotações devem definir o modo como o XML deve ser criado. Elas devem ser definidas porque o formato XML requer uma entidade raiz, ou seja, se `UsuarioResponse` tiver um campo definido por uma outra classe, não é necessário usar a anotação `XmlRootElement`.

O uso da anotação `XmlRootElement` é o principal motivo de termos dois métodos para retornar listagem de usuários: `listar` e `listarXml`. Quando usamos a especificação JSON-B, não é necessária nenhuma anotação na classe raiz, por isso podemos usar estrutura de dados como `List`, ou qualquer outra classe que implementa a interface `Collection`. Mas ao usar JAXB devemos usar a anotação, isso implica que não podemos usar listas como resposta. Essa diferença ocorre pela origem dos dois formatos de serialização, o XML não prevê a serialização de uma lista, isso deve ser feito usando um objeto que contém uma lista. Já o JSON pode serializar tanto listas quanto objetos.

```
@XmlRootElement(name = "usuarios")
@XmlAccessorType(XmlAccessType.FIELD)
```

```
public class UsuariosResponse {

    private List<UsuarioResponse> usuarios;
    // implementação com Getters e Setters
}
```

— Estamos falando de XML e JSON, mas e se por acaso eu quiser um outro formato qualquer? Eu quero definir meu próprio formato!

Essa indagação não é incomum, é bem provável que em algum momento você tenha que implementar uma funcionalidade de relatório que exportará XML, CSV, Excel, PDF e outros possíveis formatos. Em algumas soluções temos a mistura da lógica de negócios com o código necessário para exportar os dados no formato desejado. Mas existe uma forma de facilmente adicionar esse novo formato sem alterar a lógica de negócios.

A especificação JAX-RS define a interface `MessageBodyWriter`, uma implementação dessa interface será responsável por serializar objetos em um, ou mais, formatos. Para que o Quarkus encontre a implementação dessa interface, é preciso usar a anotação `Provider`. No exemplo a seguir, estamos definindo que a classe `RelatorioUsuarioWriter` deve ser usada para gerar PDF a partir de objetos `UsuariosResponse`.

```
@Provider
@Produces("application/pdf")
public class RelatorioUsuarioWriter implements
MessageBodyWriter<UsuariosResponse> {
    // implementação
}
```

Ao criarmos a classe `RelatorioUsuarioWriter`, ela poderá ser usada nas classes endpoint apenas referenciando `application/pdf` nas anotações `Produces`.

```
@Path("/usuario")
@ApplicationScoped
public class UsuarioEndpoint {
```

```
    @GET
    @Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON,
"application/pdf" })
    public UsuariosResponse listar() {
        // implementação
    }
}
```

Da mesma forma que criamos um `MessageBodyWriter`, podemos criar um `MessageBodyReader`. Este pode ser usado para associar um formato específico a um objeto. No exemplo a seguir estamos lendo `UsuariosResponse` do formato YAML.

```
@Provider
@Consumes("application/yaml")
public class RelatorioUsuarioReader implements
MessageBodyReader<UsuariosResponse> {
    // implementação
}
```

— E os cabeçalhos `Accept-Language`? Não dá pra usar com `Consumes` e `Produces`?

Em pouquíssimos casos será necessário tratar o cabeçalho `Accept-Language` em uma API. Na maioria das vezes ele será tratado automaticamente pelo framework usado para renderização de páginas pelo servidor. Mas no caso anterior, estamos tratando de uma funcionalidade típica de uma API que deve tratar internacionalização.

Só que não temos uma funcionalidade específica tratando de internacionalização no JAX-RS, por isso devemos tratar como um cabeçalho. Isso significa que ele pode ser acessado usando a anotação `HeaderParam` do JAX-RS, como no exemplo adiante, ou pode ser gerado pela implementação de `MessageBodyWriter` através do parâmetro `httpHeaders` do método `writeTo`.

```
@Path("/usuario")
@ApplicationScoped
```

```
public class UsuarioEndpoint {
    @GET
    @Path("relatorio")
    @Produces("application/pdf")
    public UsuariosResponse listar(@HeaderParam("Accept-Language") String
acceptLanguage) {
        // implementação
    }
}
```

4. Apresentando as credenciais

Para implementar uma camada de segurança usando Basic Authentication ou JWT temos sempre duas escolhas. Ou podemos tentar criar nossa própria solução ou podemos usar soluções prontas de frameworks. Criar soluções próprias só é recomendado em casos onde há especificidades no requisito, e deve ser feito com muito cuidado para não inserir falhas de segurança na sua API.

Para mostrar como configurar o HTTP Basic Authentication, vamos usar o plugin `quarkus-elytron-security-jdbc`. Com ele, é possível acessar diretamente os usuários de uma base relacional. Para configurar o plugin não é necessário nenhuma alteração em código, apenas nas configurações.

O HTTP Basic Authentication será usado como qualquer outro método de autenticação, baseando em papéis (*roles*). A seguir, podemos ver as configurações necessárias para a consulta de senha e papéis de um usuário em uma base relacional.

```
quarkus.security.jdbc.enabled=true
quarkus.security.jdbc.principal-query.sql=SELECT u.password, u.role FROM
test_user u WHERE u.username=?
quarkus.security.jdbc.principal-query.clear-password-mapper.enabled=true
quarkus.security.jdbc.principal-query.clear-password-mapper.password-
index=1
quarkus.security.jdbc.principal-query.attribute-mappings.0.index=2
quarkus.security.jdbc.principal-query.attribute-mappings.0.to=groups
```

Já para configurar a funcionalidade usando JWT também nos baseamos em um plugin, o `quarkus-smallrye-jwt`. Semelhantemente ao HTTP Basic Authentication, também não é necessária nenhuma configuração em código, apenas em arquivos de configuração.

A seguir, podemos ver que o JWT não acessa nenhuma base de dados, apenas uma chave pública e o *issuer*. A chave pública é usada para validar que o token foi encriptado com a chave privada desejada e o `issuer` deve ser o mesmo contido no token. As roles do usuário são acessadas através do campo `groups` do JWT.

```
mp.jwt.verify.publickey.location=publicKey.pem
mp.jwt.verify.issuer=https://example.com/issuer
quarkus.native.resources.includes=publicKey.pem
```

Tendo configurado método de autenticação, ou o HTTP Basic Authentication ou o JWT, a configuração das permissões é feita através das mesmas anotações `@PermitAll`, `@DeclareRoles` e `@DenyAll`. `@PermitAll` remove toda e qualquer validação de usuário, isso significa que qualquer acesso será permitido, mesmo que não tenha nenhum valor de autenticação. `@DeclareRoles` habilita a validação da autenticação baseado nos papéis que o usuário possui, não permitindo acesso sem autenticação. Já o `@DenyAll` remove todo acesso de usuário. Para exemplificar o uso, vamos demonstrar na classe `UsuarioEndpoint`.

```
import javax.annotation.security.DenyAll;
import javax.annotation.security.PermitAll;
import javax.annotation.security.RolesAllowed;
import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;

@Path("/usuario")
@DenyAll
```

```java
@ApplicationScoped
public class UsuarioEndpoint {

    @GET
    @RolesAllowed(Roles.ADMIN)
    public List<UsuarioResponse> listar() {
        // implementação
    }

    @GET
    @RolesAllowed(Roles.ADMIN)
    @Path("/{id}")
    public UsuarioResponse encontrarPorId(@PathParam("id") Long id) {
        // implementação
    }

    @Resource
    SessionContext ctx;

    @GET
    @RolesAllowed(Roles.USER)
    @Path("/me")
    public UsuarioResponse encontrarPerfil() {
        // implementação
    }

    @GET
    @RolesAllowed(Roles.ADMIN)
    @Path("/username/{username}")
    public UsuarioResponse encontrarPorUsername(@PathParam("username")
String username) {
        // implementação
    }

    @PUT
    @PermitAll
    public UsuarioResponse criar(CriarUsuarioRequest request) {
        // implementação
    }
```

```
    @Inject
    TiqueteUsuarioEndpoint tiqueteUsuario;

    @Path("{usuarioId}/tiquetes")
    public TiqueteUsuarioEndpoint tiquetesPorUsuario() {
        return tiqueteUsuario;
    }
}
```

Analisando a classe, podemos ver que, para evitar problemas de segurança, todas as operações definidas são por padrão não permitidas através do uso do `@DenyAll` na classe. Depois em cada método definimos os papéis que devem ser validados. Como estamos tratando de uma API de acesso a cadastro de usuários, todas as operações são permitidas apenas para usuários administradores e somente a operação de criação de usuário é permitida para usuários não autenticados.

No caso do endpoint `/usuario/me` será necessário acessar as informações do usuário autenticado. Para isso podemos acessar usando as informações da sessão `SessionContext`.

5. Colocando valores na requisição

Como mostramos no capítulo anterior, alguns métodos HTTP requerem um corpo na mensagem. Esse corpo será transformado automaticamente pelo JAX-RS em um POJO, e para isso precisamos primeiro criar o POJO. Ele é uma classe Java muito simples com getters e setters, ou seja, para cada campo da classe temos que criar um método com o prefixo `get` e um método com o prefixo `set`.

Para exemplificar, vamos criar o POJO da requisição de criação de usuário. Ele deverá se chamar `CriarUsuarioRequest` e ter os campos `username`, `email`, `password` e `roles`.

```java
@XmlRootElement(name = "usuario")
@XmlAccessorType(XmlAccessType.FIELD)
public class CriarUsuarioRequest {
    private String username;
    private String email;
    private String password;
    private Set<String> roles;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public Set<String> getRoles() {
        return roles;
    }

    public void setRoles(Set<String> roles) {
        this.roles = roles;
    }
```

```
    // Implemente hashCode, equals e toString usando sua IDE.
}
```

Observe nesse código que um POJO deve seguir um padrão estrito de nomenclatura, os getters e setters devem começar com `get` e `set`, respectivamente, e também devem ter a primeira letra em maiúsculo, assim `username` vira `getUsername` e `setUsername`. As anotações `XmlRootElement` e `XmlAccessorType` são usadas para definir a forma como o POJO será serializado em XML usando JAXB. Quando ele é serializado em JSON não é necessária nenhuma anotação.

Para definir o corpo da mensagem, a criação do POJO é a principal atividade. Com ele já definido, basta o utilizar como parâmetro de um método anotado com `POST` ou `PUT` e que define a anotação `Consumes`.

Segue o exemplo que já usamos anteriormente.

```
@PUT
@PermitAll
public UsuarioResponse criar(CriarUsuarioRequest request) {
    // implementação
}
```

6. Adicionando parâmetros de busca

Os parâmetros de busca são essenciais em consultas. Com eles, podemos definir filtros e outras configurações que podem alterar o resultado da busca.

Para se ter acesso aos parâmetros de busca em um método, usando o JAX-RS, deve-se usar a anotação `QueryParam`. No exemplo a seguir, adicionamos alguns filtros e uma ordenação a listagem de usuário, sendo que `Ordem` é um enumerador.

```
@GET
@RolesAllowed(Roles.ADMIN)
public List<UsuarioResponse> listar(@QueryParam("username") String
```

```
filtroUsername,
                                    @QueryParam("email") String
filtroEmail,
                                    @QueryParam("ativo")
@DefaultValue("true") boolean ativo,
                                    @QueryParam("ordem") Ordem ordem) {
    // implementação
}
```

Quando um dos parâmetros não é especificado na requisição, o valor recebido é `null`. Se fizermos uma requisição `/usuário?username=ve&order=EMAIL` teremos os valores de `filtroUsername` igual a `ve`, `ordem` igual a `EMAIL` e `ativo` igual a `true`. Os valores retornados dependerão da implementação desejada. No exemplo, um filtro de nome de usuário com valor `ve` pode significar que devem ser retornados todos os usuários com nome de usuário contendo `ve`. O mesmo acontecerá para email e o valor de ativo, mas `Ordem` alterará a ordenação da resposta. No caso ela, deve ser ordenada pelo email do usuário.

O parâmetro `ativo` foi declarado propositadamente como um tipo primitivo, que não aceita valores `null`, para demonstrar o uso da anotação `DefaultValue`. Quando usamos tipos primitivos, devemos obrigatoriamente usar essa anotação para não ocorrerem erros. Essa anotação aceita um valor em String que será transformado para o tipo definido. Para que essa anotação seja usada em classes que não tenham um tipo primitivo correspondente, a classe deve implementar os seguintes requisitos:

- Ter um construtor que aceita uma String como argumento;
- Ter um método estático nomeado `valueOf` ou `fromString` que aceita um único argumento do tipo String (veja o exemplo de `Integer.valueOf(String)` e `java.util.UUID.fromString(String)`);
- Ter uma implementação extensão do JAX-RS `ParamConverterProvider` registrada que retorne uma instância de `ParamConverter` capaz de criar o objeto a partir de uma String.

- Ser do tipo `List<T>`, `Set<T>` ou `SortedSet<T>`, onde `T` satisfaz os itens (1) ou (2) acima. O resultado é uma coleção somente leitura.

Uma das formas de satisfazer o item (3) é usar o Java CDI como no exemplo a seguir. `ApplicationScoped` significa que apenas uma instância da classe será usada em toda a aplicação e ela será disponibilizada para quem requerer via Java CDI.

```
@ApplicationScoped
public class DateParamConverterProvider implements ParamConverterProvider
{

    @Override
    public <T> ParamConverter<T> getConverter(Class<T> rawType, Type
baseType, Annotation[] annotations) {

        if (rawType == null) {
            return null;
        }
        if (rawType.equals(Date.class)) {
            return (ParamConverter<T>)new DateParamConverter(new
SimpleDateFormat("yyyy-MM-dd"));
        }
        return null;
    }
}
```

7. Escrevendo a resposta

Nesse ponto já deve estar claro que para retornar um valor na resposta da mensagem basta definir um POJO e retornar no método. Sim, essa é a forma padrão do JAX-RS, mas existe outra forma que cobrirá casos especiais onde parâmetros da resposta devem ser alterados. Esses casos especiais veremos mais à frente, por enquanto vamos demonstrar apenas os dois formatos de retornar um corpo da mensagem.

O primeiro caso especial é quando precisamos alterar valores específicos da resposta. Nesse caso o método deve retornar um objeto do tipo `Response`. Esse objeto é criado usando um builder exposto na mesma classe, como podemos ver a seguir. Nesse builder é possível definir uma entidade que será o corpo da resposta e outros valores. O segundo caso especial é quando não há resposta, e nesse caso devemos usar um método sem um tipo de retorno.

```
@PUT
@PermitAll
public Response criar(CriarUsuarioRequest request) {
    // implementação
    return Response.created(new URI(String.format("/usuario/%s",
usuario.getId())))
                   .entity(usuario)
                   .cacheControl(cacheControl)
                   .build();
}

@DELETE
@Path("/{id}")
@RolesAllowed("ADMIN")
public void criar(@PathParam("id") String param) {
    // implementação
    return;
}
```

Como vimos em **Decidindo o formato**, corpo da mensagem é retornado de acordo com os valores da anotação `Produces`, que foi omitida no exemplo.

8. Informando o tipo de resposta

Nós falamos no capítulo anterior que o tipo da resposta é definido pelo código do status da mensagem. Por padrão quando um POJO é retornado, o código usado é o 200, que significa sucesso. Mas e se desejarmos usar outros códigos?

Definir o código de status é uma tarefa que foge ao caso base do JAX-RS. Para isso, é preciso abdicar de retornar um POJO e retornar um objeto do tipo `Response` (da mesma forma que usamos anteriormente). O `Response` já tem alguns métodos com status predefinidos, mas qualquer valor pode ser usado no construtor. O status é um valor aberto dentro do protocolo. Uma API pode definir status próprios desde que sigam os intervalos das centenas como vimos no capítulo anterior.

A seguir vemos três tipos diferentes de criar uma resposta. Em `atualizar`, usamos um método que instancia um builder sem especificar o código de status, porém em `criar` usamos diretamente o código, mas em ambos temos o mesmo tipo de objeto, que pode ser usado da mesma forma. Em `apagar` já temos uma nova construção, pois o status `204` implica que não há nenhum retorno na mensagem, isso significa que o método não deve retornar nada.

```
@POST
@Path("/{id}")
@Consumes({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
@Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
public Response atualizar(AtualizarUsuarioRequest request) {
    // implementação
    return Response.ok() // 200
                   .entity(usuario)
                   .build();
}

@POST
@Path("/")
@Consumes({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
@Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
public Response criar(CriarUsuarioRequest request) {
    // implementação
    return Response.status(Status.CREATED) // 201
                   .entity(usuario)
                   .build();
}
```

```
@DELETE
@Path("/{id}")
public void apagar(@PathParam("id") String id) {
    // implementação
    return; // 204
}
```

9. Tratando erros

Existem dois tipos de erros em qualquer software, os esperados e os inesperados. Erros esperados são aqueles que devem ser identificados por validações de entrada. Já erros inesperados são aqueles que acontecem devido a uma instabilidade no sistema (por exemplo, problemas de concorrência ou falha de comunicação com a base de dados). Os JAX-RS prevê os dois tipos de tratamento como podemos ver.

Erros não devem ser retornados usando um objeto do tipo `Response`, eles estão além da regra de negócios, por isso devem ser tratados à parte. Caso a verificação faça parte explicitamente da regra de negócios, deve-se usar uma exception que pode ser tanto própria quanto uma definida pela especificação (uma subclasse de `WebApplicationException`).

Quando falamos de validação de parâmetros a resposta nem sempre está no JAX-RS, mas na especificação Jakarta Bean Validation (https://beanvalidation.org/). Com ela, é possível definir validações sem poluir o código. Por exemplo, no código a seguir, `CriarUsuarioRequest` deve ser validado antes de chamar o método.

```
@POST
@Path("/")
public Response criar(@Valid CriarUsuarioRequest request) {
    // implementação
    return Response.status(Status.CREATED) // 201
                   .entity(usuario)
```

```
                .build();
}
```

A validação de `CriarUsuarioRequest` será feita conforme as anotações definidas na própria classe. A implementação do Quarkus para o Jakarta Bean Validation valida os campos corretamente e gera uma resposta de erro com todas as mensagens de erros que podem ser usadas como mensagens exibidas para o usuário final.

```
public class CriarUsuarioRequest {

    @Email
    @NotBlank(message = "email é um campo obrigatório")
    private String email;

    @Size(min = 4, max = 15, message = "username deve ter tamanho [{min}, {max}]")
    @NotBlank(message = "username é um campo obrigatório")
    @Pattern(regexp = "^[a-zA-Z][a-zA-Z0-9]+$", message = "\"username\" deve iniciar com uma letra e só deve aceitar letras e números")
    private String username;

    // outros campos + getters e setters
}
```

— Tá bom, mas com validações eu só cubro os erros esperados. Como faço com os erros inesperados? Sei que é uma falha de segurança grave exibir uma `StackTrace` !

Sim! Em todo servidor web os erros devem ser devidamente tratados para não serem expostos via API. No JAX-RS, o tratamento de erros inesperados é feito através de uma implementação da interface `ExceptionMapper`. Essa implementação deve transformar uma `Exception` em `Response`. No exemplo a seguir tratamos todas as exceções do tipo `WebApplicationException`, mas essa abordagem ainda não cobre todos as possíveis exceções. O recomendado seria também tratar qualquer `RuntimeException`.

```
@Provider
public class ErrorMessageMapper implements
ExceptionMapper<WebApplicationException> {

    @Override
    public Response toResponse(WebApplicationException exception) {
        // some code
    }

}
```

STACKTRACE

Algumas linguagens de programação definem objetos para armazenar o contexto de execução. Isso é feito com o intuito de ajudar na depuração de erros de execução. Esses objetos são chamados de *Stacktrace*, e contêm a informação do rastreio da execução no momento em que o erro ocorre. Normalmente uma stacktrace contém os dados da pilha de execução (*stack* significa pilha e *trace* significa vestígio ou traço em inglês), como o método que está sendo executado.

Uma stacktrace pode parecer inofensiva, mas ela revela quais bibliotecas e frameworks seu sistema usa. Com essa informação, um possível ofensor pode explorar falhas conhecidas. Por isso proteja sua stacktrace! Nunca deixe que sua API essa informação.

10. Usando o Cache

No capítulo anterior, exploramos como o Cache pode ser usado para evitar desperdício de recursos. Quando o número de requisições aumenta, o uso de CPU e memória também aumenta, e para combater isso nada melhor do que diminuir o número de requisições. Isso é o que chamamos de cache, quando o cliente claramente informa quando é preciso retornar o valor ou não.

A especificação JAX-RS prevê o uso de mecanismos de cache, mas ele se dá apenas com a construção da resposta. O armazenamento da informação deve ser feito no cliente. A construção da resposta pode ser feita usando o método `evaluatePreconditions` disponível na requisição. A requisição pode ser acessada usando a anotação `Context` em um argumento do método. No código a seguir vemos como é simples o uso com mapas.

```
@GET
@Path("/{id}")
@Produces(MediaType.APPLICATION_JSON)
public Uni<Response> encontrarUsuarioPorId(@PathParam("id") String id,
@Context Request request) {
    ResponseBuilder resp =
request.evaluatePreconditions(lastModifiedMap.getOrDefault(id, EPOCH),

etagMap.getOrDefault(id, DEFAULT_ETAG));
    if (Objects.nonNull(resp)) {
        return Uni.createFrom().item(resp.build());
    } else {
        Usuario usuario = // encontrar usuário
        return Response.ok()
                        .cacheControl(CACHE_CONTROL)
                        .tag(etagMap.put(id, new
EntityTag(Integer.toHexString(usuario.hashCode()))))
                        .lastModified(lastModifiedMap.put(id,
usuario.getLastUpdate()))
                        .entity(usuario)
                        .build();
    }
}
```

A implementação do cache deve ser feita com muita cautela, podemos cair em duas tentações muito sutis que tornam o mecanismo inútil.

A primeira tentação é querer consultar os valores de `ETag` e `Last-Modified` da base de dados. Ao acessar um banco adicionamos uma latência que é equivalente à latência de requisitar o valor pedido

pela requisição. Se cedermos a essa tentação, no melhor caso, todo o ganho de performance usando cache será desperdiçado acessando a base de dados. No pior caso, teremos dois acessos ao banco para retornar um valor.

A segunda tentação é armazenar todos os valores possíveis de cache. Para entidades dinâmicas, o cache deve ser implementado apenas para recursos que são demandados com alta frequência, e mesmo dentre esses recursos, apenas para elementos altamente demandados. Vamos imaginar uma rede social hipotética. Armazenar perfil de usuários é importante porque alguns são demandados constantemente, mas armazenar perfis de baixa visibilidade não é importante. Se cairmos na tentação de armazenar os perfis de baixa visibilidade o gasto com memória pode ser elevado.

Implementar qualquer mecanismo de cache é complexo e requer muito planejamento e avaliação. Ele deve ser implementado com cautela e testes para os recursos mais demandados. Se investimos esforço para otimizar pontos que não são os gargalos do nosso sistema, estamos desperdiçando esforços.

11. Criando um canal full duplex de comunicação

Apesar de o WebSocket ser uma extensão do protocolo HTTP, ele apresenta algumas limitações. Para podermos ver essas limitações precisamos conhecer a API JavaScript para abrir uma conexão WebSocket. No código a seguir, uma conexão é aberta, uma mensagem é enviada e depois a conexão é fechada.

```
let socket = new WebSocket("ws://meuservico.com.br/canal/tiquete-101");
socket.onopen = () => {
    socket.send("Oi! Estou fechando a conexão");
    socket.close();
};
```

A conexão é aberta na primeira linha do código. Ao contrário da API `XMLHttpRequest` (usada para fazer requisições HTTP), não é possível configurar os cabeçalhos antes de abrir a conexão. E a API

WebSocket (*https://developer.mozilla.org/pt-BR/docs/Web/API/WebSocket*) não nos dá possibilidade de configurar os cabeçalhos. A autenticação pode ser feita usando o HTTP Basic Authentication através da URL (usar `ws://usuario:senha@meuservico.com.br/canal/tiquete-101`), mas JWT não é possível por não poder configurar o cabeçalho `Authorization` . Mas é através do WebSocket que conseguimos a troca assíncrona de mensagens entre cliente e servidor. A qualquer momento, o servidor pode enviar mensagens para o cliente.

Para implementar a conexão WebSocket precisamos definir uma URL. No nosso caso, vamos criar um canal de comunicação para cada tíquete do nosso sistema. Nesse canal é preciso implementar alguns métodos para tratar os seguintes eventos: `OnOpen` , `OnMessage` , `OnClose` e `OnError` . Cada evento desse terá associada uma sessão. No objeto da sessão é possível acessar o canal de comunicação com o cliente, em que é possível enviar mensagens de texto, objetos ou dados binários. Para enviar objetos é preciso definir um encoder. No exemplo a seguir, vemos um caso simples em que cada estado é comunicado ao cliente.

```
@ApplicationScoped
@ServerEndpoint(value = "/Canal/{tiquete}",
                encoders = {JsonEncoder.class},
                decoders = {JsonDecoder.class})
public class CanalSocket {

    @OnOpen
    public void onOpen(Session session, @PathParam("tiquete") String
tiquete) {
        // implementação
    }

    @OnClose
    public void OnClose(Session session, @PathParam("tiquete") String
tiquete) {
        // implementação
    }
```

```
    @OnError
    public void onError(Session session, Throwable throwable) {
        // implementação
    }

    @OnMessage
    public void onMessage(Mensagem mensagem, @PathParam("tiquete") String
tiquete) {
        // implementação
    }

}
```

A conexão WebSocket é feita a partir de um caminho, por isso é possível usar as anotações `PathParam` e `QueryParam`. Outras funcionalidades podem ser construídas através da conexão. Por exemplo, caso seja preciso uma funcionalidade de autenticação e autorização, o token JWT pode ser enviado como primeira mensagem, mas isso será específico da aplicação.

5.4 Como se aprofundar?

Neste capítulo vimos como implementar o protocolo HTTP usando o Quarkus. O framework é muito mais rico do que estamos descrevendo, mas para entendê-lo é preciso conhecer sua documentação. Como falamos no começo do capítulo, o framework valoriza muito a experiência de desenvolvimento, por isso visitar o site pode ser uma grata surpresa desde que leia textos em inglês. Se você não consegue ler textos em inglês, procure por blogs e tutoriais em vídeo.

Nessa pequena introdução não tratamos de outros pontos como Configurações, Validações, ORM (Object Relational Mapper) e clientes HTTP.

Estilos arquiteturais usando HTTP

Nesta parte vamos discutir alguns estilos arquiteturais usando HTTP. Primeiro vamos definir o que é um estilo arquitetural e depois apresentar o REST como o principal estilo arquitetural do HTTP.

Depois vamos apresentar os outros estilos que podemos encontrar no mercado.

Capítulo 6
Estilos arquiteturais e REST

Nos últimos capítulos, falamos muito do protocolo HTTP, mas agora é o momento de explorar outras visões. Você consegue definir o que é REST? Ou o que é RESTful?

Nos últimos 10 anos API REST entrou no vocabulário de qualquer desenvolvedor web, mas definir esse termo não é trivial. Sempre que encontro um desenvolvedor que diz que sabe desenvolver APIs REST pergunto o que significa isso, e na grande maioria das vezes a resposta ou é incorreta, ou é imprecisa. Por isso vamos explorar essas outras 4 letras e ver o que elas podem nos trazer de informação.

Estilos arquiteturais

O termo REST surgiu de uma pesquisa feita por Roy Thomas Fielding durante o seu doutorado. Em sua dissertação *Architectural Styles and the Design of Network-based Software Architectures* (https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm) Fielding se propõe a analisar os estilos arquiteturais para softwares baseado em rede. Software baseado em rede vai muito além da internet como a conhecemos. Nessa definição se encaixam a computação distribuída, sistemas operacionais distribuídos e

qualquer sistema descentralizado que usa a internet para se comunicar. Como é um campo bem amplo, ele faz uma análise dos estilos arquiteturais existentes em todos os contextos e depois foca seu estudo em avaliar os desafios de um software de aplicação, que em sua definição significa software voltados para negócios.

— Por que eu preciso saber disso ao invés de apenas conhecer o REST?

REST é o estilo arquitetural proposto por Fielding para ser usado em software de aplicação. Conhecer a origem do REST vai nos ajudar a compreender quando e como usar ele corretamente. Ao vê-lo como um estilo arquitetural, podemos também conhecer as motivações que fizeram com que ele fosse proposto e isso nos dá argumentos para defender o seu uso, ou mesmo avaliar o uso de um outro estilo. Para começar precisamos primeiro entender um pouco sobre o que é arquitetura de software.

Uma arquitetura de software é uma abstração de um software em execução. Ela pode ser composta de muitos níveis: podemos olhar para os componentes que interagem entre si pela rede, como o cliente e o servidor, ou podemos olhar um componente e como ele instancia seus subcomponentes em execução. Quando analisamos software em rede, vamos considerar os componentes como caixas pretas. Não nos interessa o funcionamento interno dele, apenas como eles interagem entre si. Devemos fazer perguntas como qual protocolo eles usam, como eles descobrem outros serviços em execução e como eles respondem ao aumento repentino da quantidade de requisições.

Uma arquitetura é composta de elementos, configurações, propriedades, estilos e visualizações.

Elementos são a unidade básica de uma arquitetura. Podem ser componentes, conectores, dados. Os componentes são os serviços que temos em execução no nosso sistema, sendo que cada componente terá uma, ou mais, interface com o restante da

arquitetura. Os conectores são os mecanismos que mediam a comunicação entre os componentes. Os dados são toda a informação que é trafegada dentro do sistema - não se refere somente a bases de dados, mas qualquer informação.

Com esses elementos podemos descrever a arquitetura de qualquer sistema. Se temos um serviço com uma API HTTP, podemos dizer que esse serviço é um componente com um conector HTTP. E se ele compartilha a mesma base de dados com outro serviço? Então podemos definir a base de dados como um elemento; a conexão com a base, como um conector e os dados armazenado também fazem parte da arquitetura. Mas se o serviço usa um middleware Pub/Sub, podemos dizer que esse middleware é um componente, a comunicação com ele é um conector e os dados trocados também fazem parte da arquitetura.

As **configurações** são as informações necessárias pelos componentes para atingir seus objetivos arquiteturais. O endereço IP de um servidor é uma configuração do cliente, o tópico de um Pub/Sub é uma configuração de um componente.

As **propriedades** são as características intrínsecas de um componente para atingir seus objetivos arquiteturais. As possíveis propriedades podem ser:

- O componente não armazena estado interno de cada conexão, permitindo ser replicado;
- O componente possui um endpoint `/healthy` que informará se todas as dependências dele estão funcionando;
- O componente responde sincronamente a requisições HTTP.

Os **estilos** são um conjunto de propriedades que são comuns ao sistema como um todo. Existem alguns estilos arquiteturais já catalogados e para compor nosso sistema podemos escolher apenas algumas propriedades. Não é necessário que um sistema tenha todas as propriedades de um estilo arquitetural, isso deve ser considerado antes do desenvolvimento. Um sistema pode se

adequar a mais de um estilo dependendo das suas necessidades de negócio.

Para se escolher um estilo e as propriedades deles, devemos sempre fazer perguntas como: quais são os ganhos e limitações que uma propriedade podem trazer?

As **visualizações** são qualquer representação gráfica que procura demonstrar características de um sistema. Elas podem exibir o sistema como um todo, ou apenas determinada característica. Podemos fazer um desenho apenas dos elementos que interagem por um determinado protocolo, ou podemos fazer um desenho do fluxo de um determinado dado e como ele é alterado por vários componentes.

Uma arquitetura pode ser classificada como *As Is* e *To Be* (**como é** e **como será**, em tradução livre). Uma arquitetura *as is* é aquela que descreve o sistema como ele está implementado. Ela é uma referência do que existe e deve ser o mais próximo dele. Já uma arquitetura *to be* representa o desejo de como o sistema deve ser. Ela deve representar o estado futuro desejado, não tendo qualquer pretensão de representar a realidade atual. Dadas essas duas arquiteturas, o time de pessoas arquitetas e desenvolvedoras pode elaborar um plano de migração, e planejar como transformar o que se tem hoje para o que se deseja.

Uma arquitetura existe para refletir objetivos de negócio. O sistema tem que responder a uma grande demanda de requisições? O sistema tem que responder imediatamente? Ou a resposta pode ser postergada? Quem responde a essas perguntas é o negócio que o sistema resolve. Um sistema e-commerce para uma pequena loja será totalmente diferente de um sistema e-commerce para os grandes nomes do mercado.

6.1 Estilos arquiteturais para sistemas baseado em rede

Na sua dissertação, antes de propor o REST, Fielding vai trazer uma revisão dos estilos que existiam no mercado. Ele classifica esses estilos como:

- *Pipe and Filter* (PF) e *Uniform Pipe and Filter* (UPF)
- *Cache*
- *Stateless*
- Cliente-Servidor
- Código sob demanda
- Integração baseada em eventos
- Objetos distribuídos
- Middleware

Conhecer esses estilos é útil para aumentar o nosso repertório arquitetural, alguns podem parecer óbvios, mas quando falamos de arquitetura sempre devemos focar nas qualidades de cada estilo. Além de conhecer o estilo, devemos saber descrevê-lo com suas características, vantagens e desvantagens.

Pipe and Filter e Uniform Pipe and Filter

Pipe and Filter é um sistema em que o dado flui de um componente para o outro, criando um fluxo de dados. Hoje usamos o termo *Stream* para nomear esse fluxo. Um componente lê um dado, transforma-o e pode enviá-lo, ou não, para outro sistema. Falamos de Uniform Pipe and Filter quando todos os componentes têm a mesma interface de comunicação.

Quando vamos descrever um sistema Pipe and Filter temos que detalhar todos os tipos de dados, as transformações que cada componente faz no dado e as interfaces de comunicação entre cada componente. Quando vamos descrever um sistema Uniform Pipe and

Filter, assumimos que todas as interfaces são iguais, ou seja, todos têm o mesmo tipo de conectores.

O Sistema Pipe and Filter mais conhecido que existe é o bash. Cada componente é um programa e a saída de um pode ser a entrada de outro. No exemplo a seguir, a saída de `echo` é usada na entrada do `base64` e é salva no arquivo *credenciais*.

```
echo "admin:1234" | base64 > credenciais
```

Se formos mais específicos, na verdade, o bash é um Uniform Pipe and Filter, pois todas as interfaces são em formato texto. Existem arquiteturas Pipe and Filter para sistemas web, que são conhecidos como Sistemas Orientado a Eventos. Neles, os serviços se comunicam pelo envio de mensagens assíncronas, o que quer dizer que quem envia a mensagem não tem conhecimento sobre quem recebe e não espera uma resposta para ela. Por ser uma arquitetura que dispensa o uso do protocolo HTTP, está fora do escopo deste livro.

Cache

Sistemas cacheados são usados para evitar processamento desnecessário. Ele pode ser feito em vários níveis: podemos falar em cache de requisições HTTP ou cache de informações da base de dados. Podemos também criar caches de informações geradas, por exemplo, há sistemas que criam cache do HTML gerado a partir de dados das bases, assim o HTML é alterado somente quando a informação da base é alterada.

Todo cache implica em replicação. Muitas vezes a replicação de dado é uma característica não desejada, porém nesse caso é desejada para se atingir uma melhor performance considerável.

Stateless

Stateless significa "sem estado" em tradução livre. Ou seja, são componentes que não possuem estado interno. Isso pode ter vários

significados: podem ser sistemas que somente transformam dados; ou sistemas que, tendo uma base de dados, não armazenam informações de requisições anteriores, então, cada nova requisição deve conter todos os dados importantes para ela.

Componentes stateless são bastante úteis porque podem ser facilmente escalados. Se não há estado interno, qualquer instância pode responde a qualquer requisição.

Cliente-Servidor

Esse é o estilo mais conhecido de qualquer desenvolvedor ou desenvolvedora web. Um sistema Cliente-Servidor é formado por ao menos dois componentes com responsabilidades bem definidas. Cabe ao cliente elaborar requisições e cabe ao servidor respondê-las. Um servidor nunca atua ativamente abrindo a conexão, esta sempre é iniciada pelo cliente.

Código sob demanda

Em um sistema que implementa o estilo de código sob demanda, o cliente tem acesso a um conjunto de recursos e ao código para interpretá-lo. Além de prover esses recursos o servidor pode também prover o código para interpretá-lo.

Isso pode parecer muito abstrato, mas essa é a base de qualquer framework front-end. A aplicação front-end é fornecida pelo servidor através de arquivos JavaScript, e ela pode ser cacheada no cliente. A maior vantagem de um sistema Código sob demanda é que o sistema pode ser atualizado sem atualizar a versão do cliente.

Integração baseada em eventos

Em uma integração baseada em eventos, o conector não identifica somente dois pontos, ela pode integrar múltiplos componentes. Em vez de um componente enviar uma mensagem a apenas outro componente, ele envia uma mensagem como *broadcast*, então ela

pode ser recebida por quem se interessar. Sistemas baseados em eventos, muitas vezes, precisam de um componente para controlar as mensagens, e esse componente pode ou não armazenar essas mensagens.

Objetos distribuídos

Em um sistema com objetos distribuídos, um componente pode interagir com outro como se estivesse usando um objeto localmente. Os dados são armazenados tanto de um lado quanto de outro, há um estado interno no objeto e há uma interface de acesso ao objeto. Objetos distribuídos são usados para isolar o processamento de dados. Um componente pode controlar sessão e dados em outro sistema.

Middleware

Middleware são componentes que atuam para resolver problemas de integração. Eles não precisam encapsular nenhuma lógica de negócios do sistema, podem servir apenas para troca de mensagens ou armazenamento de informações. Eles uniformizam interfaces e protocolos entre vários componentes. Usando um middleware, não há a necessidade de um componente conhecer o outro. Isso pode ser abstraído por um tópico ou alguma outra interface lógica.

6.2 REST, um estilo arquitetural

O REST é proposto como um estilo arquitetural e pode ser descrito como a conjunção de vários dos estilos descritos acima. Para definilo, Fielding começa com um estilo vazio, sem nenhuma propriedade, mas identificando as forças que o movem a escolher algumas propriedades. Assim ele começa a construir um estilo híbrido, mesclando propriedades de outros estilos.

O primeiro estilo a ser escolhido é o obvio: qualquer sistema web é cliente-servidor, cada componente tem sua responsabilidade bem definida.

A segunda propriedade de um sistema REST é que o servidor deve ser *stateless*. Toda informação de estado deve ser armazenada no cliente e a requisição deve conter toda a informação necessária para ser respondida. Por ser stateless, o servidor não precisa armazenar informações de sessão. Essa propriedade permite a um servidor responder um grande número de clientes.

A terceira propriedade é que as requisições podem ser cacheadas. Criando-se uma interface uniforme e onde o recurso é identificado pela URI, pode-se adicionar uma camada de cache tanto no cliente quanto como em um componente de cache entre o cliente e o servidor. O cache vai evitar processamento desnecessário, diminuindo o número de requisições entre vários componentes do sistema.

A quarta propriedade é a uma interface uniforme. APIs REST são construídas levando em conta que os recursos podem ser identificados através da URI e a ação identificada pelo verbo HTTP.

A quinta propriedade é que um sistema REST pode ser composto por várias camadas. Ao se desenhar uma interface URI uniforme, ela pode ser distribuída entre vários componentes. Podemos ter componentes de cache, componentes específicos para recursos e componentes para recursos estáticos.

Por fim, a última propriedade de um sistema REST é que o código pode ser provido sob demanda. Uma API REST pode prover, além de recursos, arquivos contendo a lógica de negócios. Essa propriedade permite termos o que hoje conhecemos como *Single Page Application* e *Progressive Web Applications*.

Com essas propriedades do REST podemos supor outras propriedades. A primeira delas é que o REST é um estilo arquitetural síncrono composto de requisição e resposta. Essa propriedade é

derivada do próprio protocolo HTTP pois toda requisição HTTP é síncrona com uma resposta associada. Outra característica é que uma troca de informação só ocorre entre dois componentes da arquitetura, não há o *broadcasting* de informações, isto é, uma mesma informação não pode ser enviada para vários componentes.

Como construir uma API REST?

A primeira decisão que devemos fazer para construir uma API REST é definir quais são os recursos controlados por ela. Originalmente, o HTTP foi proposto para controlar documentos, logo, imagina-se que eles têm uma correspondência entre URI e arquivo. Porém, quando falamos de REST, criamos outra abstração. A URI não representa um documento, mas algum recurso que é definido pela própria API. A URI se torna cidadã de primeira classe em uma API REST, ela vai definir quem é o recurso.

Se considerarmos uma URI como uma coleção de tokens separados pelo caractere barra, `/`, podemos catalogar cada token por sua função. Alguns podem caracterizar o tipo do recurso, outros, o identificador do recurso e, por fim, a ação. Ao darmos nome a um recurso, podemos já supor algumas URIs associadas a ele.

Além dos tokens, os verbos são o segundo elemento mais importante de uma API REST, eles vão definir qual ação pode ser catalogada para uma determinada URI. Não existe nenhuma definição entre um verbo e uma ação, essa é uma decisão de quem vai implementar a API. Mas existem algumas diretrizes disponíveis na internet. Como exemplo, podemos ver a definição de operações do guideline da Microsoft (*https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design#define-api-operations-in-terms-of-http-methods*).

Verbo	Descrição
GET	Recupera a representação do recurso identificado pela URI. O corpo da resposta contém os detalhes do

Verbo	Descrição
	recurso requisitado.
POST	Cria um novo recurso através na URI especificada. O corpo da requisição provê os detalhes do novo recurso. Note que o POST pode também ser usado para disparar operações que na verdade não criam um novo recurso.
PUT	Cria ou atualiza um recurso especificado pela URI. O corpo da requisição especifica o recurso a ser criado ou atualizado.
PATCH	Atualiza parcialmente o recurso. O corpo da requisição especifica o conjunto de alterações a serem aplicadas no recurso.
DELETE	Remove o recurso especificado pela URI.

Para os verbos `GET`, `PATCH` e `DELETE` há consenso sobre o uso. Mas sobre o uso do `POST` e `PUT`, podemos encontrar uma grande variedade de recomendações, até algumas que entram em conflito. Qual deve atualizar e qual deve criar? Não temos essa resposta, pois isso vai muito do contexto da API. O importante é manter uma uniformidade. Se você usa sempre o `PUT` para criar um recurso, mantenha a consistência em todo o sistema. Para facilitar no consenso, antes do desenvolvimento do projeto, crie um próprio guideline em consenso com sua equipe e documente suas escolhas.

Agora vamos usar o nosso exemplo do capítulo anterior para exemplificar? Tínhamos um software de gestão de tíquetes, então nosso primeiro recurso é o `tiquete`. Tíquetes possuem criador (`criador`) e responsável (`resp`), todos esses são usuários (`usuario`) do sistema. Tíquetes podem pertencer a um Sprint (`sprint`). Então vamos listar algumas possíveis operações que podemos criar:

Verbo	URI	Descrição
GET	/tiquete	Retorna todos os tíquetes cadastrados no sistema.
POST	/tiquete	Cria um novo tíquete no sistema.
GET	/tiquete/:id:	Retorna as informações do tíquete identificado pelo identificador `id`.
GET	/tiquete/autor/:autorId:	Retorna todos os tíquetes criados pelo autor identificado por `autorId`.
GET	/tiquete/responsavel/:respId:	Retorna todos os tíquetes em que o responsável é identificado por `respId`.
PUT	/tiquete/:id:	Atualiza as informações do tíquete identificado pelo identificador `id`.
GET	/usuario/:id:/tiquete	Retorna todos os tíquetes cadastrados no sistema onde o usuário identificado por `id` é autor ou responsável.
POST	/sprint	Cria um novo sprint.
PUT	/sprint/:id:/tiquete	Associa tíquete ao sprint identificado por `id`.
GET	/sprint/:id:/tiquete	Retorna todos os tíquetes associados ao

Verbo	URI	Descrição
		sprint identificado por `id`.

> **SPRINT**
>
> A metodologia ágil mais comum é o Scrum, nela o Sprint é definido "**como as batidas do coração do Scrum, onde as ideias se transformam em valor**" (*https://scrumguides.org/scrum-guide.html#the-sprint*). São fases de desenvolvimento de tamanho fixo que se encadeiam. Ao termino de um sprint, outro é iniciado imediatamente. Todo sprint tem um objetivo específico. Se há a necessidade de o objetivo ser alterado, isso implica no cancelamento do sprint e o início de outro. As atividades de um sprint são escolhidas de acordo com o objetivo.

Observe que essa tabela poderia ter muitos mais itens, mas preferi me ater apenas àqueles que quero demonstrar. Os tokens podem ser usados para selecionar recursos, mas também formas de acesso, como é visto em `/tiquete/autor/:autorId:`. Tokens podem ser encadeados, criando-se uma cadeia de relações, como em `/sprint/:id:/tiquete`. Nesse caso, `/sprint/:id:` serve de filtro para a operação de listagem de tíquetes.

Observe que, se um cliente já executou a chamada a `GET /tiquete/:id`, não é necessária uma nova execução, pois já sabemos o conteúdo da resposta e esse dado pode ser cachaeado no cliente.

Uma API RESTful

Agora que conhecemos o que é o estilo REST, você pode se perguntar: mas por que sempre encontro a palavra RESTful? O que ela significa?

Quando falamos REST, estamos nos referindo ao estilo arquitetural. Quando falamos RESTful estamos nos referindo a uma implementação. Assim, uma API implementada usando esse estilo pode ser chamada de RESTful.

Ela vai diferir, por exemplo, de uma API SOAP, onde a operação não é descrita pela URI, mas pelo corpo da mensagem.

— Mas para que realmente serve uma API RESTful?

APIs REST são muito úteis, elas foram desenhadas para responder a um determinado tipo de problema: *como expor recursos para serem gerenciados por uma aplicação cliente-servidor*. Caso o seu problema não se adeque a isso, você terá muita dificuldade em modelar sua API.

Como vimos que API RESTful tem um padrão uniforme de URI, elas se tornam fácil de usar por um desenvolvedor. O desenvolvedor pode supor que alterando o verbo, pode-se encontrar uma nova funcionalidade da API. Esse padrão de URI também possibilita o cache de recursos, evitando o uso desnecessário do servidor.

O que não é REST

Uma das práticas que tenho levado durante minha carreira é nunca supor nada ao conversar com outros profissionais. Na prática, isso quer dizer que, quando alguém me fala que trabalha dando manutenção ou implementando uma API REST, a primeira coisa que faço é perguntar:

- O que você entende por API?
- O que você entende por REST?

Essa abordagem me trouxe um dado muito interessante sobre a visão que as pessoas têm sobre o próprio trabalho. Muitos não sabem definir ao certo os termos com que eles trabalham. Na maioria dos casos, as pessoas não sabem me dizer o que é uma API, ou se uma biblioteca pode ser uma API também. Em outros casos,

as pessoas têm uma definição completamente errada. Elas identificam algumas características e afirmam que aquilo é uma definição. É nessa última categoria que o REST se enquadra.

Creio que 70% das pessoas a quem já fiz essa pergunta me responderam a mesma coisa: *Uma API REST é uma API HTTP que responde JSON.* Ora essa definição é parcialmente correta e completamente errada. Vamos analisá-la?

Podemos extrair dela algumas afirmações verdadeiras, como *Uma API REST é uma API HTTP*, ou *Uma API REST responde JSON*. As duas são verdadeiras, mas se fizermos o exercício de inverter as frases vemos que elas são errôneas. Podemos afirmar claramente que *nem toda API HTTP é REST* e *nem toda API HTTP que responde JSON é REST*.

Você pode até reler todo este capítulo e a tese de Fielding para tentar procurar alguma menção a JSON e não a encontrará. Isso porque o padrão REST não especifica o formato do corpo da requisição e da resposta. Ele se preocupa somente com o design da URI e o uso dos verbos HTTP.

6.3 Conclusão

Estilos arquiteturais são a força delineadora dos nossos sistemas. Eles devem estar intrinsecamente alinhados com a solução de negócio que desejamos resolver. Podemos escolher um ou mais estilos arquiteturais para compor o nosso sistema, mas para isso precisamos primeiro conhecer os estilos que existem. Quando limitamos o nosso repertório de estilos arquiteturais, limitamos a nossa possibilidade de resolução de problemas.

É muito comum escolhermos o estilo arquitetural errado para implementarmos nossa solução. Há casos em que somos levados pelo estilo arquitetural da moda e acreditamos que ele se adequa ao nosso problema. Quando isso acontece, a solução começa a ficar demasiadamente complexa. Ao primeiro sinal de complexidade desnecessária ao modelar seu sistema, primeiro se pergunte se não está faltando algo no modelo. Seria um novo recurso? Seria uma dependência entre recursos? Se você não encontrar resposta, ou ainda a modelagem está muito complexa, pode ser que você esteja com um problema que pode ser respondido por outro estilo arquitetural.

Capítulo 7
Outros estilos para HTTP

O protocolo HTTP não é usado somente para criar APIs REST, existem outros estilos arquiteturais que podem ser implementados através dele. Neste capítulo vamos falar de GraphQL, (g)RPC e SOAP. Nosso foco aqui não será discutir esses estilos arquiteturais a fundo, mas apresentá-los para que você possa responder a perguntas como:

- Quando é melhor usar gRPC?
- Quando devo usar GraphQL?
- SOAP? Alguém ainda usa?

7.1 RPC, gRPC e SOAP?

gRPC é bem conhecido, sendo um tema recorrente em conferências de desenvolvimento de software, mas não é o mais conhecido deles. No gráfico a seguir podemos ver as estatísticas de busca no Google dos termos gRPC, RPC e SOAP desde 2004. Percebe-se que o ápice da popularidade dessas tecnologias foi em 2004 e nessa época RPC era buscado quatro vezes mais que SOAP. Hoje o SOAP e RPC ainda continuam na dianteira e seguidos pelo gRPC, que só surgiu em 2016. Isso acontece porque essas tecnologias são bastante utilizadas em sistemas antigos ainda em atividade, afinal, o surgimento de novas tecnologias não implica na automática aposentadoria de tecnologias antigas - elas existirão enquanto existirem sistemas as utilizando.

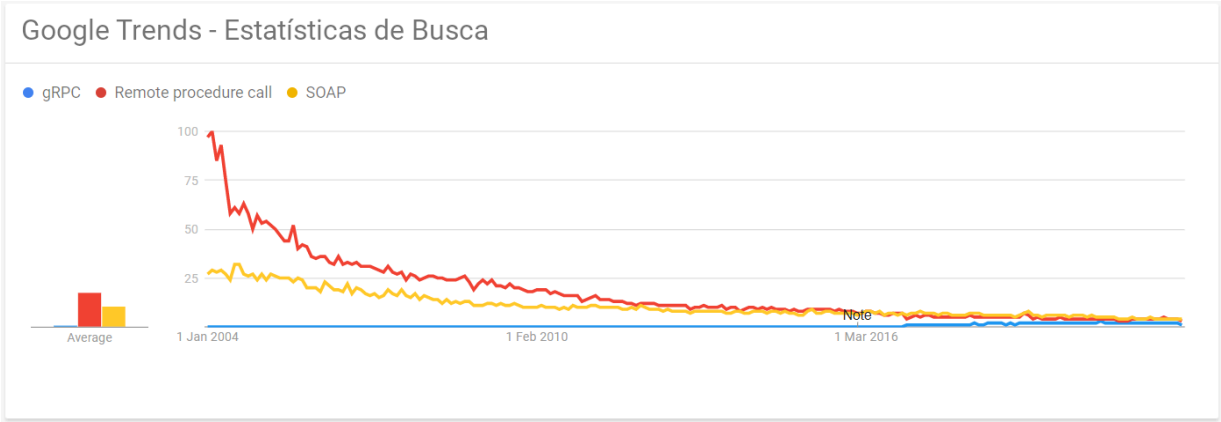


Figura 7.1: Estatísticas de Busca

Apesar de ser uma sopa de letrinhas, estamos falando de apenas um estilo arquitetural. gRPC e SOAP no fundo são implementações RPC, o *Remote Procedure Call*. O RPC começou sua história em 1976 (*https://tools.ietf.org/html/rfc707*). Ele parte do pressuposto de que cliente e servidor podem se comunicar como se estivessem dentro do mesmo processo.

Em outras palavras, RPC é um modelo de programação que pode ser implementado de várias maneiras, mas nunca a rede, o protocolo de comunicação e o método de serialização fazem parte das preocupações dos desenvolvedores. Para um desenvolvedor usando RCP, basta conhecer as interfaces de código e trabalhar como se estivesse utilizando um serviço dentro do mesmo processo.

Essas abstrações são possíveis desde que exista um framework que gere código automaticamente. Assim, por exemplo, ao usar o gRPC a pessoa desenvolvedora deve definir uma interface usando Protocol Buffers e o código usado tanto no cliente quanto no servidor é gerado pelo framework, sendo possível de utilizar em diversas linguagens (C#, C++, Dart, GO, Java, Kotlin, Node, Objective-C, PHP, Python e Ruby).

— Mas qual é a grande vantagem de seu usar RPC?

Para responder a essa pergunta, vamos considerar o uso do RCP em oposição ao REST. Quando usamos uma API REST temos que nos preocupar com vários detalhes do protocolo HTTP que são desnecessários em certos contextos.

Por exemplo, se estamos falando de um tratamento de exceção no REST, precisamos discutir qual *status code* devemos usar e o formato da resposta. Mas se estamos falando de RPC só precisamos definir qual `Exception` o método pode arremessar. As definições de qual *status code* ou qual caminho usar são delegadas ao framework, ficando ao desenvolvedor o trabalho de definir uma biblioteca que será chamada pela própria linguagem.

Apesar da facilidade de se usar o RPC como uma chamada local, quem está desenvolvendo não deve ceder à tentação de acreditar que é realmente uma chamada local. Toda chamada RPC envolve dois serviços e se devem considerar os erros possíveis para qualquer chamada através da rede. Podemos ter o servidor fora do ar, podemos ter um problema de rede que torne a conexão instável, ou o servidor pode receber processar a mensagem, mas a resposta não chegar ao cliente. Esses são erros que devem ser considerados no desenvolvimento de qualquer sistema envolvendo a internet.

7.2 gRPC, RPC de roupa nova

Podemos dizer que o gRPC é o RPC de roupa nova. Em sua concepção, nada muda das antigas implementações, mas ele usa uma série de tecnologias novas que permitem baixa latência, alta taxa de transferência de dados e comunicação bidirecional.

Desenvolvido pelo Google desde 2015, o gRPC é um framework RPC que usa Protocol Buffers (*https://developers.google.com/protocol-buffers*), mais conhecido como protobuf, como linguagem para definição de interfaces e HTTP/2 como protocolo. Protobuf é uma

biblioteca de serialização de dados implementada pelo Google, que está disponível em várias linguagens. Uma mensagem gera um dado binário otimizado para minimizar seu tempo de serialização e transmissão.

Apesar de ser baseado em HTTP, o gRPC não é baseado em recursos. Ao fazer o design de uma API gRPC, não devemos focar em recursos, mas em serviços. Uma API gRPC é orientada a serviços. Vamos demonstrar através de um exemplo? Vamos imaginar nosso serviço de tíquetes: primeiro precisamos definir os tipos de dados que serão utilizados como requisições e respostas e depois quais os serviços que estarão disponíveis.

Para nosso serviço, vamos definir os seguintes objetos `EncontraTiquetePorUsuario` e `Tiquete`. Observe que `Tiquete` é composto de vários outros objetos.

```
message EncontraTiquetePorUsuario {
    string username = 1;
}

message Usuario {
    string id = 1;
    string username = 2;
    string email = 3;
    string nome = 4;
}

message Tiquete {
    string id = 1;
    string titulo = 2;
    string descricao = 3;
    Usuario responsavel = 6;
    Usuario criador = 7;
}
```

— O que é importante em uma mensagem protobuf?

Um campo dentro de uma mensagem protobuf terá ao menos três informações: o tipo, o nome e o número identificador. Ao contrário dos outros métodos de serialização, o nome no protobuf não é importante, ele é descartado durante a serialização.

Um campo protobuf é identificado pelo tipo e por seu número identificador, esses nunca podem ser alterados. Uma mensagem protobuf aceita novas versões dela mesma. Caso um dos lados da comunicação adicione ou remova um campo, o outro lado conseguirá se comunicar sem nenhum impacto.

Como próximo passo, precisamos definir o serviço. No arquivo protobuf, não precisamos implementar o serviço, apenas definir a assinatura de cada procedimento. No nosso exemplo simples, vamos definir um serviço de busca de tíquetes que pode responder imediatamente ou criar um fluxo com os dados atualizados do tíquete.

```
service TiqueteService {
    rpc getTiquetePorUsario(EncontraTiquetePorUsuario) returns (Tiquete)
{}
    rpc subscribeTiquetePorUsuario(EncontraTiquetePorUsuario) returns
(Tiquete Weather) {}
}
```

Como um serviço gRPC é basicamente um objeto, os padrões a serem seguidos são bem conhecidos, mas precisamos ter alguns cuidados especiais. Ao se implementar tanto um cliente quando um servidor gRPC, precisamos ter em mente que a conexão pode ser instável, o que implicará que devemos nos preocupar com tratar a rede com retentativas ou, simplesmente, assumir que o serviço está indisponível momentaneamente.

Se executarmos a requisição para `getTiquetePorUsario`, será enviada uma requisição HTTP/2 com o método POST e o caminho composto por `/TiqueteService/getTiquetePorUsario`, com o corpo da mensagem composto do conteúdo de `EncontraTiquetePorUsuario`, que é binário.

7.3 SOAP, o irmão mais velho do gRPC

O SOAP é, com certeza, o irmão mais velho do gRPC. Talvez você já tenha ouvido falar de SOAP, mas muitos desenvolvedores nunca viram um serviço em SOAP em execução.

— Se o SOAP é antigo, por que devo conhecê-lo?

SOAP foi um dos padrões mais comum para comunicação entre serviços nos anos 2000, mas ainda existem muitos serviços implementados usando SOAP. Como não podemos subestimar o poder do legado, é provável que em algum momento da sua carreira você vai encontrar algum serviço legado em SOAP.

— Mas qual a diferença do SOAP para o gRPC?

Diferentemente do gRPC, o SOAP parte da premissa básica de que o serviço já está disponível quando se implementa um cliente. *O que isso significa?* Quando vamos criar um cliente é preciso prover um documento XML chamado WSDL (*Web Services Description Language*). Esse documento será bem similar ao arquivo `proto` usado no gRPC, mas esse documento sempre é compartilhado pelo servidor.

Se tivermos um end-point `http://tiquetes.com.br/soap` implementando SOAP, posso concluir que consigo acessar o arquivo WSDL, enviando uma requisição GET em `http://tiquetes.com.br/soap?wsdl` .

SOAP usa HTTP e XML com um end-point para cada serviço. Diferentemente do REST, para o SOAP o verbo HTTP tem pouca relevância. O GET, por exemplo, é somente usado para adquirir a documentação, e para todos os outros serviços é usado somente o POST. Isso acontece porque, como cada end-point encapsula um serviço com várias possíveis chamadas, uma requisição deve ter um corpo especificando a funcionalidade desejada assim como os parâmetros.

No exemplo a seguir, podemos ver uma requisição para listar todos os tíquetes e sua respectiva resposta.

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:cxf="http://cxf.acme.org/">
    <soapenv:Header />
    <soapenv:Body>
        <cxf:listarTodosTiquetes />
    </soapenv:Body>
</soapenv:Envelope>
```

Observe que o SOAP não usa detalhes do protocolo HTTP e com isso acaba recriando no corpo das mensagens informações que poderiam ser encapsuladas no cabeçalho da requisição HTTP. Tanto requisição quanto resposta têm o conteúdo similar usando XML, todos devem ser encapsulado por `Envelope` e `Body`.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <ns1:listarTodosTiquetesResponse
xmlns:ns1="http://server.soap.roadmap.dev.com.br/">
            <return>
                <tiquete>
                    <id>6942</id>
                    <titulo>Erro ao criar Tíquete</titulo>
                    <descricao>A API de criação de tíquetes retorna erro
500 quando nenhuma tag é associada.</descricao>
                    <projeto>PRJ-001</projeto>
                    <epico>CRIAR-TIQUETE</epico>
                    <tags>
                        <tag>Java</tag>
                        <tag>REST</tag>
                        <tag>CRUD</tag>
                    </tags>
                </tiquete>
            </return>
        </ns1:listarTodosTiquetesResponse>
```

```
    </soap:Body>
</soap:Envelope>
```

O comportamento do SOAP é muito similar ao do gRPC, ambos possuem o mesmo estilo arquitetura (RPC), mas com elementos diferentes. O gRPC, por ter como requisito baixa latência, usa *protocol buffers* para serialização e HTTP/2, já o SOAP, por ser mais antigo, se limita ao XML.

7.4 GraphQL, a API flexível

O GraphQL é um estilo arquitetural que permite a geração de consultas complexas usando HTTP. Dentre todos os estilos que vimos até agora, temos em comum que sempre conhecemos os dados enviados na requisição e na resposta. Esse é o pressuposto do REST e das implementações de RCP. Mas o GraphQL foi criado para tentar responder a uma outra demanda: como resolver consultas dinâmicas sem precisar de várias requisições.

GraphQL é bastante usado para aplicações que têm vários tipos de clientes. Vamos supor que o nosso gerenciador de requisições de mudanças fosse implementado como aplicação web e mobile, e para cada cliente temos informações diferentes a serem exibidas.

Usando REST ou SOAP o formato da requisição não pode ser alterado em tempo de execução. Seriam necessários dois endpoints diferentes ou uma inconsistência no modelo. Mas com o GraphQL podemos definir quais campos e relacionamentos desejamos retornar.

Vamos imaginar, no nosso serviço de requisições de mudanças, que podemos cadastrar tíquetes e fazer buscas. A interface GraphQL será exposta juntamente à API REST, mas somente vai expor duas chamadas. A primeira listará todas as issues e a segunda buscará uma issue pelo título.

Ao se implementar esse serviço usando GraphQL, temos a definição do schema de API a seguir. Temos a definição dos tipos e das as consultas a serem executadas (*Query*), mas isso não implica que esse tipo será retornado por uma requisição.

Em uma chamada GraphQL, além da requisição, deve ser enviado qual é o modelo de dados desejado, composto pelo campo que deverá ser exibido.

```
"Query root"
type Query {
  encontrarTiquete(titulo: String): [Tiquete]
  listarTodosTiquetes: [Tiquete]
}

type Tiquete {
  id: Int!
  titulo: String!
  descricao: String
  projeto: String!
  epico: String
  tags: [String]
  criador: Usuario!
  responsavel: Usuario
  observadores: [Usuario]
}

type Usuario {
  id: Int!
  username: String!
  email: String!
  nome: String
  sobrenome: String
}
```

No exemplo a seguir podemos ver uma chamada que faz uma busca por um tíquete e seleciona quais campos devem retornar. O cliente seleciona os campos que são relevantes para ele, evitando o tráfego

indevido de dados, reduzindo acessos e concentrando todas as informações em uma só requisição.

```
POST /graphql HTTP/1.1
Host: localhost:8080
User-Agent: curl/7.80.0
Accept: */*
Content-Type: application/json
Content-Length: 231

{"query":"{\r\n    encontrarTiquete(titulo: \"Erro ao criar Tíquetes\")
{\r\n        id\r\n        titulo\r\n        projeto,\r\n        criador
{\r\n            id\r\n            username\r\n        }\r\n
}\r\n}","variables":{}}

HTTP/1.1 200 OK
content-type: application/graphql+json; charset=UTF-8
content-length: 136

{"data":{"encontrarTiquete":[{"id":6942,"titulo":"Erro ao criar
Tíquete","projeto":"PRJ-001","criador":{"id":124,"username":"vepo"}}]}}
```

O código-fonte desse exemplo pode ser encontrado no endereço *https://github.com/vepo/quarkus-graphql*.

O GraphQL é um estilo arquitetural extremamente flexível que deve ser usado quando o formado do dado não é homogêneo entre todos os clientes da API. Ele não é concorrente com o REST e pode ser implementado em uma API REST através de um único end-point, compartilhando até a mesma base de código.

7.5 Conclusão

Os estilos vistos neste capitulo devem ser usados sempre que existir a necessidade. Nem todo problema pode ser resolvido com uma

única solução, por isso devemos expandir nosso repertório de soluções a aprender quando elas podem ser usadas.

SOAP ainda é bastante usado em soluções e não creio que ela vá desaparecer no curto prazo. Existem sistemas e empresas que usam ESB (*Enterprise Service Bus*, ou barramento de serviços corporativos em tradução livre) para agrupar e disponibilizar vários serviços usando SOAP.

gRCP é bastante útil quando falamos de comunicação entre serviços. Talvez não seja uma boa ideia usá-lo para uma API aberta, o REST possui ferramentas mais úteis para esse fim. Mas ele pode prover uma abordagem fácil e prática para que dois times possam debater uma interface e cada time desenvolver a sua parte confiando no código gerado a partir dos arquivos `proto`.

GraphQL por sua vez é muito útil quando temos vários tipos de clientes. Se cada cliente tem demandas diferentes, a melhor decisão é que o cliente decida o que quer consumir.

Capítulo 8
Próximos passos

Quando estou lendo um livro e ele termina, eu sempre me pergunto: *quais são os meus próximos passos?* Você também pode se fazer essa pergunta. Uma pessoa que trabalha com desenvolvimento de software é alguém que está sempre aprendendo e a indústria está em constante movimento. Por isso, este livro não se focou em mostrar o último framework do momento e como ele funciona exatamente, porque ele vai mudar, mas o que você aprendeu nessas páginas ainda permanecerá útil e atual por muitos anos.

Falamos bastante de HTTP, por isso creio que você deve começar a modelar APIs. Discuta com o seu time os modelos que foram propostos aqui. Tente encontrar formas melhores de resolver os problemas que seu projeto atual tem. Todo software evolui, e com o que vimos, você pode propor melhorias no que você tem hoje.

Se quiser reforçar os conceitos de REST, modele uma API para algum modelo de negócios. Em uma das mentorias que fiz, a pessoa modelou uma API para consultórios de psicologia. Depois de modelar a API, ela implementou o serviço usando as tecnologias mais recentes e esse exercício foi bastante útil para que ela pudesse conseguir uma realocação no mercado. Na maioria das empresas não temos liberdade de usar as tecnologias que queremos, mas podemos estudá-las em casa.

Outro ponto muito importante para desenvolvedores back-end é conhecer como uma base de dados funciona. Existem bases relacionais, o famoso SQL, e bases não relacionais (NoSQL). Em muitos projetos, elas já vêm configurada com um framework ORM, dos quais o mais comum é o Hibernate. Conhecer o framework ORM e como sua base funciona também vai habilitar você a escrever software de qualidade. Não subestime os problemas que um ORM mal modelado pode trazer ao sistema.

Existe um outro tipo de comunicação para serviços back-end de que não falamos neste livro, os serviços de mensageria. Muitos falam do Apache Kafka, ele é o mais complexo de todos. Se você deseja ampliar seu vocabulário, procure estudar o que é uma Event-Driven Architecture. Às vezes, o Apache Kafka é um canhão para matar uma mosca, você consegue o mesmo resultado com serviços mais simples fornecidos pelos provedores cloud. Mas o mais importante é conhecer como usar um sistema dele e quando usar.

Como próximos passos, você poderia treinar tudo aquilo que aprendeu neste livro tentando criar APIs melhores e usando ao máximo as potencialidades do protocolo HTTP. Meu objetivo foi detalhar algumas das informações que podem ser úteis para você, desde sobre como o protocolo HTTP pode ser usado, o que é uma API RESTful e outros padrões arquiteturais. A computação e o marcado vão evoluir bastante nos próximos 10 anos, mas é muito provável que este livro continue atual porque essas tecnologias estão por aí faz muitos anos e vão continuar por muitos outros.